

尽享5小时完整视频教程！跟着数十万人的Python导师学Python！

PEARSON

“笨办法”学Python（第3版）

Learn PYTHON the HARD WAY

THIRD EDITION

[美] Zed A. Shaw 著

王巍巍 译



人民邮电出版社

POSTS & TELECOM PRESS

VISIT…

LANZAROTE

Caliente.com

Table of Contents

笨办法学Python	1.1
序言	1.2
前言	1.3
简介	1.4
练习0.安装和准备	1.5
练习1.第一个程序	1.6
练习2.注释和井号“#”	1.7
练习3.数字和数学计算	1.8
练习4.变量和命名	1.9
练习5.更多的变量和打印	1.10
练习6.字符串和文本	1.11
练习7.更多的打印（输出）	1.12
练习8.打印，打印	1.13
练习9.打印，打印，打印	1.14
练习10.那是什么?	1.15
练习11.提问	1.16
练习12.提示别人	1.17
练习13.参数，解包，变量	1.18
练习14.提示和传递	1.19
练习15.读文件	1.20
练习16.读写文件	1.21
练习17.更多文件操作	1.22
练习18.命名，变量，代码，函数	1.23
练习19.函数和变量	1.24
练习20.函数和文件	1.25
练习21.函数的返回值	1.26
练习22.到目前为止你学到了什么？	1.27
练习23.阅读代码	1.28
练习24.更多的练习	1.29
练习25.更多更多的练习	1.30

练习26.恭喜你，可以进行一次考试了 1.31
练习27.记住逻辑 1.32
练习28.布尔表达式 1.33
练习29.IF 语句 1.34
练习30.Else 和 If 1.35
练习31.做出决定 1.36
练习32.循环和列表 1.37
练习33.while循环 1.38
练习34.访问列表元素 1.39
练习35.分支和函数 1.40
练习36.设计和调试 1.41
练习37.复习符号 1.42
练习38.列表操作 1.43
练习39.字典,可爱的字典 1.44
练习40.模块, 类和对象 1.45
练习41.学会说面向对象 1.46
练习42.对象、类、以及从属关系 1.47
练习43.基本的面向对象的分析和设计 1.48
练习44.继承Vs.包含 1.49
练习45.你来制作一个游戏 1.50
练习46.项目骨架 1.51
练习47.自动化测试 1.52
练习48.更复杂的用户输入 1.53
练习49.写代码语句 1.54
练习50.你的第一个网站 1.55
练习51.从浏览器获取输入 1.56
练习52.开始你的web游戏 1.57
来自老程序员的建议 1.58
下一步 1.59
附录A：命令行教程 1.60
附录A-简介 1.60.1
附录A-练习1：安装 1.60.2
附录A-练习2：路径, 文件夹, 名录 (pwd) 1.60.3
附录A-练习3：如果你迷路了 1.60.4

附录A-练习4：创建一个路径 (mkdir)	1.60.5
附录A-练习5：改变当前路径 (cd)	1.60.6
附录A-练习6：列出当前路径 (ls)	1.60.7
附录A-练习7：删除路径 (rmdir)	1.60.8
附录A-练习8：目录切换(pushd, popd)	1.60.9
附录A-练习9：生成一个空文件(Touch, New-Item)	1.60.10
附录A-练习10：复制文件 (cp)	1.60.11
附录A-练习11：移动文件 (mv)	1.60.12
附录A-练习12：查看文件 (less, MORE)	1.60.13
附录A-练习13：输出文件 (cat)	1.60.14
附录A-练习14：删除文件 (rm)	1.60.15
附录A-练习15:退出命令行 (exit)	1.60.16
附录A-下一步	1.60.17

笨办法学 python

> 译者：gastlygem
>
> 来源：LPTHW

这本书指导你在Python中通过练习和记忆等技巧慢慢建设和建立技能,然后应用它们解决越来越困难的问题。在这本书的最后，你需要拥有必要的工具开始进行更多复杂程序的学习。我喜欢告诉大家，我的书带给你们“编程黑带”。意思是说你知道的基础知识足够现在就开始学习编程。

序言

这本书面向没有太多基础的人群去学习Python，在国外有很多的粉丝。

> 英文原文地址：http://learnpythonthehardway.org/book/

欢迎来到用笨办法学python的第三版。你可以访问合作站点http://learnpythonthehardway.org/，在那里你可以购买这本书的电子版下载和纸质书。您也可以在http://learnpythonthehardway.org/book/ 阅读这本书的免费HTML版本。

> 本书中文出处：http://flyouting.gitbooks.io/learn-python-the-hard-way-cn/content/

前言

这本简单书的目的是让你起步编程。虽然书名说是“笨办法”,但其实并非如此. 所谓的“笨办法”是指本书教授的方式。这本书的教学方式就是按照我告诉你的方式去做一系列的练习，目的是通过重复练习掌握一种技能。这对于一些什么都不知道的初学者，在理解更复杂的科目之前获取基本能力是很有效的方法。这种方法适用于一切领域，从武术到音乐甚至基本的数学和阅读技巧。

这本书指导你在Python中通过练习和记忆等技巧慢慢建设和建立技能,然后应用它们解决越来越困难的问题。在这本书的最后，你需要拥有必要的工具开始进行更多复杂程序的学习。我喜欢告诉大家，我的书带给你们“编程黑带”。意思是说你知道的基础知识足够现在就开始学习编程。

如果你认真学习，利用好你的时间，并学会这些技能，你就可以学习编程。

致谢

我要感谢Angela在这本书前两个版本里对我的帮助。没有她的帮助，我可能根本不能完成这本书。她帮我完成了第一份书稿的编辑工作，同时在我写这本书的过程一直很支持我。

我还想感谢Greg Newman帮我制作了封面，Brian Shumate为我完成了早期的网 站设计，以及阅读了这本书并给我反馈帮助我修正这本书的所有人。

谢谢。---

简介

笨办法更简单

在这本书的帮助下，你将通过非常简单的练习学会一门编程语言。做练习 是每个程序员的必经之路：

> 1. 做每一道习题
> 2. 一字不差地写出每一个程序
> 3. 让程序运行起来

就是这样,刚开始可能会非常难，但你要坚持下去。如果你通读了这本书，并且每晚花一两个小时做习题，你可以为自己读下一本关于Python的编程书籍打下良好的基础。这本书不会在一夜之间把你变成一个程序员，但是它会帮你掌握学习编程的最基本的方法。

这本书的目的是教会你作为编程新手所需的三种最重要的技能：读和写、注重细节、发现不同。

读和写

如果你连打字都成问题的话，那你学习编程也会成问题。尤其如果你连程序源代码中的那些奇怪字符都打不出来的话，就根本别提编程了。没有这种基本技能的话，你将连最基本的软件工作原理都难以学会。

输入代码样例并让他们运行起来能帮你记住各种符号的名字并对它们熟悉起来，这个过程也会让你对编程语言更加熟悉。

注重细节

区分好坏程序员的最重要的一个技能就是对于细节的注重程度。事实上这是任何行业区分好坏的标准。你必须关注你工作中任何一个微小的细节，否则你的工作成果将缺乏重要的元素。以编程来讲，这样你得到的结果只能是毛病多多难以使用的软件。

通过将本书中的例子一字不差地打出来，你将通过实践训练自己，让自己集中精力到你作品的细节上面。

发现不同

程序员长年累月的工作会培养出一个重要技能，那就是对于不同点的区分能力。有经验的程序员拿着两份仅有细微不同的程序，可以立即指出里边的不同点来。程序员甚至造出工具来让这件事更加容易，不过我们不会用到这些工具。你要先用笨办法训练自己的大脑，等你具备一些相关能力的时候才可以使用这些工具。

在你做每一个习题的时候，你一定会写错东西。这是不可避免的，甚至有经验的程序员也会偶尔出点错。你的任务是对比你写过的东西和正确的答案，并将所有的不同点都改正。这个过程可以训练你关注自己的错误，bugs以及其他的一些问题。

不要复制-粘贴

你必须自己手动将每个练习打出来。复制粘贴会让这些练习变得毫无意义。这些习题的目的是训练你的双手和大脑思维，让你有能力读代码、写代码、观察代码。如果你复制粘贴的话，那你就是在欺骗自己，而且这些练习也将失去效果。

使用书中包含的视频

《笨办法学Python》一书中包含超过5小时的教学视频。对于每一个练习都有一个视频，或者是示范这个练习，或者是给出一些完成练习的提示。使用视频的最佳方式是首先尝试不使用它们完成练习，然后通过视频回顾所学，或者是在你被问题卡住的时候使用视频。这将慢慢使你通过视频来学习编程和构建你直接理解代码的技能。坚持下去,慢慢的你将不需要书中视频或任何学习编程的视频。你可以只看你所需要的信息。

对于坚持练习的一点建议

在你通过这本书学习编程时，我正在学习弹吉他。我每天至少练习2个小时，至少花一个小时练习音阶、和声、和琶音，剩下的时间用来学习音乐理论和歌曲演奏以及训练听力等。有时我一天会花8个小时来练习吉他，因为我觉得这是一件有趣的事情。对我来说，要学好一样东西，重复的练习是必不可少的。就算这天个人状态很差，或者说学习的课题实在太难，你也不必介意，只要坚持尝试，总有一天困难会变得容易，枯燥也会变得有趣了。

在我写笨办法学Python和笨办法学Ruby之间的那段时间，我发现了绘画这个有意思的事情。我在39岁的时候爱上了视觉艺术，并且花费每天的时间来学习它，就像我学习吉他，音乐和编程一样。我收集教学材料的用书，按照书上讲的做，每天练习绘画，并且专注于享受学习的过程。我不是一个“艺术家”，但是现在我可以说我会绘画。我在这本书中教给你我用到学习艺术上的相同方法。如果你把问题分解成小的练习课，并且每天完成他们，你就可以做任何事情了。如果你把精力集中在慢慢改进，享受学习的过程，那么你一定会受益，不管你之前有多么擅长它。

在你通过这本书学习编程的过程中要记住一点，就是“万事开头难”，对于有价值的事情尤其如此。也许你是一个害怕失败的人，一碰到困难就想放弃。也许你是一个缺乏自律的人，一碰到“无聊”的事情就不想上手。也许因为有人夸你“有天分”而让你自视甚高，不愿意做这些看上去很笨拙的事情，怕有负你”神童”的称号。也许你太过激进，把自己跟有20多年经验的编程老手相比，让自己失去了信心。

不管是什么原因，你一定要坚持下去。如果你碰到做不出来的加分习题，或者碰到一节看不懂的习题，你可以暂时跳过去，过一阵子回来再看。只要坚持下去，你总会弄懂的。一开始你可能什么都看不懂。这会让你感觉很不舒服，就像学习人类的自然语言一样。你会发现很难记住一些单词和特殊符号的用法，而且会经常感到很迷茫，直到有一天，忽然一下子你会觉得豁然开朗，以前不明白的东西忽然就明白了。如果你坚持练习下去，坚持探索他们，你最终会学会这些东西的。也许你不会成为一个编程大师，但你至少会明白程序是怎么工作的。

如果你放弃的话，你会失去达到这个程度的机会。你会在第一次碰到不明白的东西时(几乎是所有的东西)放弃。如果你坚持尝试，坚持写习题，坚持尝试弄懂习题坚持阅读习题的话，你最终一定会明白里边的内容的。如果你通读了这本书，却还是不知道编程是怎么回事。那也没关系，至少你尝试过了。你可以说你已经尽过力但成效不佳，但至少你尝试过了。这也是一件值得你骄傲的事情。

给“小聪明”们的警告

有的学过编程的人读到这本书，可能会有一种被侮辱的感觉。其实本书中没有任何要居高临下地贬低任何人的意思。只不过是我比我面向的读者群知道的更多而已。如果你觉得自己比我聪明，然后觉得我在居高临下，那我也没办法，因为你根本就不属于我的目的读者群。

如果你觉得这本书里到处都在侮辱你的智商，那我对你有三个建议:

> 1. 别读这本书了。我不是写给你的，我是写给需要学习的人的。
> 2. 放下架子好好学。如果你认为你什么都知道，那你就很难从比你强的人身上学到什么了。
> 3. 学 Lisp 去。我听说什么都知道的人可喜爱 Lisp 了。

对于其他在这里学习的人，你们读的时候就想着我在微笑就可以了，虽然我的眼睛里还带着恶作剧的闪光。

练习0.安装和准备

这道习题并没有代码内容，它的主要目的是让你在计算机上安装好 Python。你应该尽量照着说明进行操作，例如 Mac OSX 默认已经安装了 Python 2，所以就不要在上面安装Python 3 或者别的 Python 版本了。

> **Warning:**如果你不知道怎样使用 Windows 下的 PowerShell，或者 OSX 下的 Terminal，或者 Linux下的“bash”，那你就需要学习了。我有一个免费的快速入门教程放在 http://cli.learncodethehardway.org/ 你可以快速学到PowerShell 和 Terminal 的基本用法。学完后再回来看这本书吧。

Mac OS X

你需要做下列任务来完成这个练习：

> 1. 用浏览器打开 http://www.barebones.com/products/textwrangler/ 下载并安装 `TextWrangler` 文本编辑器。
> 2. 把 `TextWrangler` (也就是你的编辑器) 放到 Dock中，以方便日后使用。
> 3. 找到你的终端程序。 搜索一下，你就会找到它。
> 4. 同样将你的终端放到Dock中
> 5. 运行你的终端程序. 这个程序看上去不怎么地。
> 6. 在 Terminal 程序里边运行 `python` 。运行的方法是输入程序的名字再敲一下回车
> 7. 键入 quit(), 回车, 就能退出python.
> 8. 这样你就应该退回到敲 `python` 前的提示界面了。如果没有的话自己研究一下为什么.
> 9. 学着使用 Terminal 创建一个目录.
> 10. 学着使用 Terminal 进入一个目录.
> 11. 使用你的编辑器在你进入的目录下建立一个文件。你将建立一个文件。使用 “Save” 或者 “Save As...” 选项，然后选择这个目录.
> 12. 使用键盘切换回到 Terminal 窗口，如果不知道怎样使用键盘切换.
> 13. 回到 Terminal，使用 `ls` 命令看到你新建的文件.

OS X: 你应该看到的结果

以下是我在自己电脑的Terminal中执行上述练习时看到的内容。和你做的结果会有一些不同，但是应该相差不多。

```
Last login: Sat Apr 24 00:56:54 on ttys001
~ $ python
Python 2.5.1 (r251:54863, Feb  6 2009, 19:02:12)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> ^D
~ $ mkdir mystuff
~ $ cd mystuff
mystuff $ ls
# ... Use TextWrangler here to edit test.txt....
mystuff $ ls
test.txt
mystuff $
```

Windows

> 1. 浏览器打开http://notepad-plus-plus.org/ 下载并安装 `notepad++` 编辑器，这个操作不需要用管理员权限。
> 2. 确定你可以方便的打开 `notepad++` ，你可以把它放到桌面或者快速启动栏，两种方式在安装的时候都可以选择。
> 3. 从开始菜单运行 `PowerShell` 程序。你可以使用开始菜单的搜索功能，输入名称后敲回车即可打开。
> 4. 为它创建一个快捷方式，放到桌面或者快速启动栏中以方便使用。
> 5. 运行你的 `PowerShell` （后面我将称呼它为 `Terminal` ）。
> 6. 在 Terminal 程序里边运行python。运行的方法是输入程序的名字再敲一下回车。
>
> ```
> 1.  如果你运行 python 发现它不存在(系统找不到python云云)。你需要访问 [](http://python.org/download)[http://python.org/download](http://python.org/download) 并且安装 Python 。
> 2.  确认你安装的是 Python 2 而不是 Python 3。
> 3.  你也可以试试 ActiveState Python，尤其是你没有管理员权限的时候。
> 4.  如果你安装好了但是 python 还是不能被识别，那你需要在 powershell 下输入并执行以下命令：
> ```

```
>     5\. 关闭并重启`powershell`，确认`python`现在可以运行。如果不行的话你可能需要重启电脑。

> 1\. 键入 quit(), 回车, 就能退出python。
> 1\. 这样你就应该退回到敲`python`前的提示界面了。如果没有的话自己研究一下为什么。
> 1\. 学着使用 Terminal 创建一个目录。
> 1\. 学着使用 Terminal 进入一个目录。
> 1\. 使用你的编辑器在你进入的目录下建立一个文件。你将建立一个文件，使用 “Save” 或者 “Save As...” 选项，然后选择这个目录。
> 1\. 使用键盘切换回到 Terminal窗口，如果不知道怎样使用键盘切换， 你一样可以上网搜索.
> 1\. 回到 Terminal，使用`ls`命令看到你新建的文件.

从现在开始，当我说到`Terminal` 或者`shell`的时候，我指的是 `PowerShell`. 推荐你也用。

> **Warning:**有时这一步你会漏掉：Windows 下装了 Python 但是没有正确配置路径。 确认你在 powershell 下输入了
```

~~~你也许需要重启 powershell 或者计算机来让路径设置生效。

Windows:你应该看到的结果
~~~

```
> python
ActivePython 2.6.5.12 (ActiveState Software Inc.) based on
Python 2.6.5 (r265:79063, Mar 20 2010, 14:22:52) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
> mkdir mystuff
> cd mystuff
... Here you would use Notepad++ to make test.txt in mystuff ...
>
> dir
 Volume in drive C is
 Volume Serial Number is 085C-7E02

 Directory of C:\Documents and Settings\you\mystuff

04.05.2010  23:32    <DIR>          .
04.05.2010  23:32    <DIR>          ..
04.05.2010  23:32                 6 test.txt
               1 File(s)              6 bytes
               2 Dir(s)  14 804 623 360 bytes free
```

如果你看到跟我的信息的不同，这仍然是正确的，但是也应该是相似的。

Linux

Linux 系统可谓五花八门，安装软件的方式也各有不同。我们假设作为 Linux 用户的你已经知道如何安装软件包了，以下是给你的操作说明:

1. 使用Linux的包管理器下载并安装`gedit`.
2. 把 gedit (也就是你的编辑器)放到窗口管理器显见的位置，以方便日后使用。

```
1.  运行 `gedit`，我们要先改掉一些愚蠢的默认设定。
2.  从 `gedit menu` 中打开 `Preferences`，选择 `Editor` 页面。
3.  将 `Tab width:` 改为 4。
4.  选择 (确认有勾选到该选项) `Insert spaces instead of tabs`。
5.  然后打开 “Automatic indentation” 选项。
6.  转到 `View` 页面，打开 “Display line numbers” 选项。
```

1. 找到`Terminal`程序。它的名字可能是`GNOME Terminal``Konsole`、 或者`xterm`。

1. 把 Terminal 也放到 Dock 里面。
2. 运行 Terminal 程序，
3. 在 Terminal 程序里边运行python。运行的方法是输入程序的名字再敲一下回车.

```
a. 如果你运行 python 发现它不存在的话，你需要安装它，而且要确认你安装的是 Python 2 而非 Pyth
on 3。
```

1. 键入 quit(), 回车, 就能退出python.

1. 这样你就应该退回到敲`python`前的提示界面了。如果没有的话自己研究一下为什么。
2. 学着使用 Terminal 创建一个目录.
3. 学着使用 Terminal 进入一个目录.
4. 使用你的编辑器在你进入的目录下建立一个文件。你将建立一个文件，使用 “Save” 或者 “Save As...” 选项，然后选择这个目录。
5. 使用键盘切换回到 Terminal窗口，如果不知道怎样使用键盘切换， 你一样可以上网搜索.
6. 回到 Terminal，使用`ls`命令看到你新建的文件.

Linux:应该看到的结果

```
$ python
Python 2.6.5 (r265:79063, Apr  1 2010, 05:28:39)
[GCC 4.4.3 20100316 (prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
$ mkdir mystuff
$ cd mystuff
# ... Use gedit here to edit test.txt ...
$ ls
test.txt
$
```

如果你看到跟我的信息的不同，这仍然是正确的，但是也应该是相似的。

可以在网上找的东西

这本书最重要的一部分是学习在网络上研究编程的课题。如果我告诉你“在网上搜索这个问题的答案”，你要做的就是使用一个搜索引擎去找到答案。我让你自己搜索而不是直接告诉你答案的原因是因为我希望当你读完我的书之后，你能成为一个独立的学习者。如果你能在晚上找到自己需要的答案，你就离不需要我更近了一步，这正是我的目标。

多亏了谷歌等搜索引擎你能很容易的找到我告诉你IDE任何东西。如果我说“网上搜索python list functions”，你应该这么样做：

> 1.浏览器打开http://google.com/
>
> 2.输入: `python list functions` 3.阅读网页上列出来的最好的答案.

给新手的告诫

你已经完成了这节练习。这个练习对你而言可能会有些难，这要根据你对自己电脑的熟悉程度。如果你觉得有难度的话，你要自己克服困难，多花点时间学习一下。因为如果你不会这些基础操作的话，编程对你来说将会更难学习。

如果有人告诉你让你在书中一些特殊的练习题处停止或者跳过一些习题，你应该忽略他们。任何试图对你隐藏知识，更甚者，让你从他们而不是通过自己的努力获得知识的人，都在试图让你依赖他们而不是自己的技能。不要听他们的，要继续做练习题，这样你才能学习如何自学。

如果有程序员告诉你让你使用 vim 或者emacs， 那你应该拒绝他们。当你成为 一个更好的程序员的时候，这些编辑器才会适合你使用。你现在需要的只是一个可以编辑文字的编辑器. 我们使用 `gedit` , `TextWrangler``Notepad++` (从现在开始我们称呼它文本编辑器)是因为它很简单，而且在不同的系统上面使用起来是一样的,就连专业程序员也会使用这些编辑器，所以对于初学而言它已经足够了。

也许有程序员会告诉你让你安装和学习 Python3。 拒绝他们， 并告诉她们 “等你电脑里的所有python 代码都支持 Python 3 了，我再试着学学吧。” 这句话足够他们忙活个十来年的了。再重复一次，不要使用Python 3。Python 3 并未广泛的应用, 如果你学习了Python2，当你需要Python3的时候，就能很容易的学会。如果你学了Python3，你仍然需要学习Python 2来完成一些事情。只要学习Python2就好，忽略别人Python3才是未来的说法。

总有一天你会听到有程序员建议你使用 Mac OSX 或者 Linux。如果他喜欢字体美观，他会告诉你让你弄台 Mac OSX 计算机，如果他们喜欢操作控制而且留了一部大胡子，他会让你安装Linux。再次说明，只要有一台手上能用的电脑就可以了。你需要的只有三样东西: 一个本文编辑器、一个命令行终端、还有 python。

最后，这节练习的准备工作的目的是帮助你在以后的练习中顺利地做到下面的这些事情：

> 1. 使用你的编辑器编写练习题，在linux上使用gedit，在OS X上使用TextWrangler，或者在windows上使用 Notepad++。
> 2. 运行你编写的习题.
> 3. 修改习题中的错误.
> 4. 重复以上步骤.

练习1.第一个程序

> **Warning:**如果你没有做练习0，说明你没有用正确的办法使用本书。你需要仔细阅读我书中提到的每一点。比如，你有没有打算用Python3来完成书中的习题？我在练习0中说过不要使用Python3；你是不是打算使用什么IDE来编辑代码？我同样说过你现在不需要这些，你只需要一个文本编辑器就够了。如果你没有阅读练习0的内容，请回过头重新阅读一下。

你应该在练习 0 中花了不少的时间，学会了如何安装文本编辑器、运行文本编辑器、以及如何运行命令行终端，而且你已经花时间熟悉了这些工具。请不要跳过前一个练习的内容直接进行下面的内容，这也是本书唯一的一次这样的警示。

将下面的内容写到一个文件中，取名为 `ex1.py` 。注意这个命名方式，Python文件最好以 `.py` 结尾。

```
print "Hello World!"
print "Hello Again"
print "I like typing this."
print "This is fun."
print 'Yay! Printing.'
print "I'd much rather you 'not'."
print 'I "said" do not touch this.'
```

如果你使用的是Mac OSX系统，你看到的应该是下面的样子：

如果你在windows上使用的Notepad++编辑器，你看到的应该是下图的样子：

如果你的编辑器跟这些图都不太一样，也没关系，是比较相似的就是正确的。当你创建文件的时候，注意以下几点：

> 1.不需要输入上面内容最左侧的行号，他们的作用是我可以在跟大家讨论某一行代码的时候，可以跟大家说，“请看第几行”。你不需要把行号输入到Python的脚本中。2.我在 `ex1.py` 的每一行开始都用到了print，他们看起来是一模一样的。每一个字符在脚本中都有它自己的角色，颜色并不重要，重要的是你输入的是正确的。

然后在命令行终端通过输入以下内容来运行这段代码： `python ex1.py`

如果你输入正确的话，你应该看到和下面图片一样的内容。如果不一样，那就是你写错了什么。不是计算机出错了，计算机没错。

你应该看到的输出

在Mac OSX的终端中，你会看到：

在windows的终端中，你会看到：

在 `python ex1.py` 命令之前，你可能会看到不同的计算机或目录名字，这不是问题，重点是你输入这个命令后，能看到和我的输出一样的结果。

如果你遇到了类似下面的错误：

```
$ python ex/ex1.py
  File "ex/ex1.py", line 3
    print "I like typing this.
                             ^
SyntaxError: EOL while scanning string literal
```

你能看懂这些错误信息是很重要的，因为之后你可能会犯更多的错误。即使是我，也犯过很多的错误。下面让我们逐行的分析报错信息:

> 1.首先我们在命令行终端输入命令来运行 `ex1.py` 脚本。2.Python告诉我们 `ex1.py` 文件的第3行有一个错误。3.然后这一行的内容被打印了出来。4.然后 Python 打印出一个 `^` (井号，caret) 符号，用来指示出错的位置。 注意到少了一个 `"` (双引号，double-quote) 符号了吗？5.最后，它打印出了一个“语法错误(SyntaxError)”告诉你究竟是什么样的错误。通常这些错误信息都比较难懂，不过你可以把错误信息复制到搜索引擎里，然后你就能看到别人也遇到过这样的错误，也许你还能在网上找到如何解决这个问题。
>
> **Warning:**如果你来自另外一个国家，而且你看到关于 ASCII 编码的错误，那就在你的 python 脚本的最上面加入这一行: `# -*- coding: utf-8 -*-` 这样你就在脚本中使用了 `unicode UTF-8` 编码，这些错误就不会出现了。

附加题

你还有 附加题 需要完成。加分习题里边的内容是供你尝试的。如果你觉得做不出来，你可以暂时跳过，过段时间再回来做。

> 1.让你的脚本多打印一行；2.让你的脚本只打印一行；3.在某行的起始位置放一个 `#` (#) 符号。它的作用是什么？自己研究一下。

从现在开始，如果我们没有遇到与这个习题不同的练习，我不会再逐个解释这些习题是怎么运行的。

> **NOTE:** `#` 号有很多的英文名字，例如：`octothorpe(八角帽)`，`pound(英镑符)`, `hash(电话的#键)`, `mesh(网)` 等。

常见问题

下面是一些学生在做习题的时候提出的一些真实问题。

Q:我可以使用IDE吗？

> 不可以，你应该像我一样使用终端，如果你不知道怎么使用终端的话，你可以阅读附录A中的命令行速成教程。

Q:如何在我的编辑器里显示不同颜色？

> 先把你的文件保存为 `.py` 结尾的文件，比如 `ex1.py` ，之后你再编辑的时候，就会有颜色区别了。

Q:我执行脚本的时候，遇到一个 `SyntaxError: invalid syntax` 报错

> 你可能想运行Python，可是你多打了一次Python,重启你的终端程序，并用正确的方法输入命令 `python ex1.py` .

Q:我遇到报错 can't open file 'ex1.py': [Errno 2] No such file or directory

> 你应该进入你文件保存的目录下。确保你执行了 `cd` 命令已进入文件目录。比如，你的文件保存在目录 `lpthw/ex1.py` 下，那你应当在执行 `python ex1.py` 之前先执行 `cd lpthw/` 。如果不明白我说的什么意思，请先通读附录A。

Q:在我的文件中，如何显示我自己国家的文字？

> 在你文件的第一行输入 `# -*- coding: utf-8 -*-` 。

Q:我的文件没有运行；我的文件运行后没有输出

> 请逐字逐句的检查你的代码文件，你应该输入 `print "Hello World!"` 而不只是 `"Hello World!"` ，检查你的文件，是不是没有 `print` ，请保证你的文件和我的一模一样。

练习2.注释和井号“#”

注释在编程中是很重要的部分。它能告诉你这段代码是干什么用的，或者用来删除一部分你暂时不需要执行的代码。下面演示的是如何在python中使用注释：

```
# A comment, this is so you can read your program later.
# Anything after the # is ignored by python.

print "I could have code like this." # and the comment after is ignored

# You can also use a comment to "disable" or comment out a piece of code:
# print "This won't run."

print "This will run."
```

从现在开始，我将使用带注释的编写代码。你要明白，不是所有的东西都有文字说明的。你的屏幕和程序可能看起来不太一样，不过，最重要的应该是你输入到文件中的内容。事实上，我可以使用任意的文本编辑器编写这些代码，并且保证他们的执行结果都是一样的。

你应该看到的结果

```
$ python ex2.py
I could have code like this.
This will run.
```

同样，我不会告诉你所有可能的终端的屏幕截图.你应该明白，上面的文字并不是你的输出结果的样子，而是在你的命令行 `$ python ...` 以及最后一个 `$` 之间的文字内容。

附加题

> 1.弄清楚”#”号的作用,并且记住它的名字。(中文为井号，英文为 octothorpe 或者 pound character)。2.打开你的 ex2.py文件，从后往前逐行检查。从最后一行开始，倒着逐个单词单词检查回去。3.有没有发现什么错误？有的话就修复它们。4.大声朗读你写的代码，把每个字符都读出来。有没有发现更多的错误呢？有的话也一样改正过来。

常见问题

Q:你确定 `#` 被称为pound character?

> 我把它叫做octothorpe是因为它是唯一一个没有国家采用，但却在每个国家使用的名字。每个国家都认为注释符的名字应该有如下特性：既是最重要的注释方法也是唯一的注释方法。对我来说，这是一个很无聊的问题，你应该将精力集中在更重要的事情上，比如学习如何编程上。

Q:如果 `#` 是注释的话，那么 `# -*- coding: utf-8 -*-` 是怎么运行的？

> Python仍然会忽略这句代码，但是它却可以作为“黑客”或者解决问题的方法来制定文件的格式。你还可以在编辑器的设置中找到其他类似的注释。

Q:为什么 `print "Hi # there."` 这句中的 `#` 没有被忽略

> 这句代码中的`#`是包含在字符串中的，字符串直到遇到下一个`"`为止，字符串里的`#`只是当做一个字母而不是注释处理。

Q:我怎样注释掉多行呢？

> 在要注释的每一行前面加上`#`

Q:我不知道如何使用我们本国的键盘输入 `#`

> 一些国家使用Alt键和其他键的组合来打印他们的语言文字。你得在网上搜索下你们国家的键盘如何输入`#`。

Q:为什么要我从后向前阅读代码

> 这其实是一种欺骗你大脑的做法，这样做能让你的大脑没有附加意义的理解每一部分代码，同时能让你正确的处理你的每一块代码。这是一个方便的捕获错误，检测错误的技术。

练习3.数字和数学计算

每一种编程语言都包含处理数字和进行数学计算的方法。不必担心，程序员经常撒谎说他们是多么牛的数学天才，其实他们根本不是。如果他们真是数学天才，他们早就去从事数学相关的行业了，而不是写写广告程序和社交网络游戏。

这节练习里有很多的数学运算符号。我们来看一遍它们都叫什么名字。你要一边写一边念出它们的名字来，直到你念烦了为止。名字如下：

```
+ plus 加号
- minus 减号
/ slash 斜杠 除法
* asterisk 星号 乘法
% percent 百分号 模除
< less-than 小于号
> greater-than 大于号
<= less-than-equal 小于等于号
>= greater-than-equal 大于等于号
```

有没有注意到以上只是些符号，没有运算操作呢？写完下面的练习代码后，再回到上面的列表，写出每个符号的作用。例如 `+` 是用来做加法运算的。

```
print "I will now count my chickens:"

print "Hens", 25 + 30 / 6
print "Roosters", 100 - 25 * 3 % 4

print "Now I will count the eggs:"

print 3 + 2 + 1 - 5 + 4 % 2 - 1 / 4 + 6

print "Is it true that 3 + 2 < 5 - 7?"

print 3 + 2 < 5 - 7

print "What is 3 + 2?", 3 + 2
print "What is 5 - 7?", 5 - 7

print "Oh, that's why it's False."

print "How about some more."

print "Is it greater?", 5 > -2
print "Is it greater or equal?", 5 >= -2
print "Is it less or equal?", 5 <= -2
```

你看到的结果

```
$ python ex3.py
I will now count my chickens:
Hens 30
Roosters 97
Now I will count the eggs:
7
Is it true that 3 + 2 < 5 - 7?
False
What is 3 + 2? 5
What is 5 - 7? -2
Oh, that's why it's False.
How about some more.
Is it greater? True
Is it greater or equal? True
Is it less or equal? False
```

附加题

> 1.使用 `#` 在代码每一行的前一行为自己写一个注释，说明这一行的作用。2.记得 `练习 0` 的内容吧？用里边的方法把 `Python` 运行起来，然后使用刚才学到的运算符号，把Python当做计算器玩玩。3.自己找个需要计算的东西，写一个 `.py` 文件把它计算出来。4.有没有发现有些计算结果是"错"的呢？计算结果只有整数，没有小数部分。研究一下这是为什么，搜索一下"浮点数(floating point number)" 是什么东西。5.使用浮点数重写一遍 `ex3.py` ，让它的计算结果更准确(提示: 20.0 是一个浮点数)。

常见问题

Q: 为什么 `%` 表示模除而不是"百分号"？

> 这个问题的答案正好就是问什么大部分程序员选择用这个运算符。平时我们把它看做一个"百分号"。在编程计算中，通常把它和 `/` 一样当做除法的运算符。 `%` 是一个不同的运算，只是用 `%` 符号来表示。

Q: `%` 如何运算的？

> 另一种说法是, "X除以Y余J." 比如, "100 除以16余数为4." `%` 计算的结果就是这个余数。

Q: 运算符的顺序是怎样的？

> 在美国我们用一个括弧来指定加减乘除的顺序。Python中也同样遵循这个规律。

Q: `/` 是如何运算的？

> 它只是舍去了小数点后面的部分，比较一下 `7.0 / 4.0` 和 `7 / 4` ,你就会明白其中的不同了。

练习4.变量和命名

你已经学会了使用print语句和算术运算。下一步你要学的是“变量”。在编程中，变量只不过是用来指代某个东西的名字。程序员通过使用变量名可以让他们的程序读起来更像英语。而且因为程序员的记性都不怎么地，变量名可以让他们更容易记住程序的内容。如果他们没有在写程序时使用好的变量名，在下一次读到原来写的代码时他们会大为头疼的。

如果你被这节习题难住了的话，记得我之前教过的：找到不同点、注意细节。

> 1. 给每一行代码加上注释，给自己解释一下这一行的作用。
> 2. 倒着读你的 `.py` 文件。
> 3. 朗读你的 `.py` 文件，将每个字符朗读出来。

```
cars = 100
space_in_a_car = 4.0
drivers = 30
passengers = 90
cars_not_driven = cars - drivers
cars_driven = drivers
carpool_capacity = cars_driven * space_in_a_car
average_passengers_per_car = passengers / cars_driven

print "There are", cars, "cars available."
print "There are only", drivers, "drivers available."
print "There will be", cars_not_driven, "empty cars today."
print "We can transport", carpool_capacity, "people today."
print "We have", passengers, "to carpool today."
print "We need to put about", average_passengers_per_car, "in each car."
```

> **NOTE:** `space_in_a_car` 中的 `_` 是下划线。你要自己学会怎样打出这个字符来。这个符号在变量里通常被用作假想的空格，用来隔开单词。

你应该看到的结果

```
$ python ex4.py
There are 100 cars available.
There are only 30 drivers available.
There will be 70 empty cars today.
We can transport 120.0 people today.
We have 90 to carpool today.
We need to put about 3 in each car.
```

附加题

当我第一次写这个程序时我犯了个错误，python 告诉我这样的错误信息：

```
Traceback (most recent call last):
  File "ex4.py", line 8, in <module>
    average_passengers_per_car = car_pool_capacity / passenger
NameError: name 'car_pool_capacity' is not defined
```

用自己的话解释一下这个错误信息，解释时记得使用行号，而且要说明原因。

更多的附加题

> 1.我在程序里用了 4.0 作为`space_in_a_car`的值，这样做有必要吗？如果只用 4 会有什么问题?2.记住 4.0 是一个`浮点数`，自己研究一下这是什么意思。浮点数是带有小数点的数字。3.在每一个变量赋值的上一行加上一行注释。4.记住`=`的名字是等于(equal)，它的作用是为东西取名。5.记住`_`是下划线字符(underscore)。6.将`python`作为计算器运行起来，就跟以前一样，不过这一次在计算过程中使用变量名来做计算，常见的变量名有 i, x, j 等等。

常见问题

Q: `=` (单等号)和`==`(双等号)之间的区别?

> `=` (单等号) 用来赋值，`==`(双等号)用来判断等号两边的值是否相等。你会在27节习题里学到这些。

Q: 我们能用 `x=100` 代替 `x = 100` 吗?

> 当然可以, 但是这种写法不好。你应该在操作符的两边加上空格，这样能提高你的代码易读性。

Q: 在打印输出的时候，怎样进行字符串拼接?

> 你可以这样做: `print "Hey %s there." % "you".` , 以后你会经常这么干。

Q: "倒着读文件"是什么意思?

> 非常简单.想象你有一个16行代码的文件 。从第16行开始读，并和我的代码的第16行进行比较。然后对第15行代码重复上面的操作，直到你倒序的读完整个文件。

Q: 为什么用 `4.0` 作为 `space_in_a_car` 的值?

> 它的主要目的就是引出什么是浮点数。看看附加题部分。

练习5.更多的变量和打印

我们现在要输入更多的变量并且把它们打印出来。这次我们将使用一个叫“格式化字符串(format string)”的东西. 每一次你使用 `"` 把一些文本引用起来， 你就建立了一个字符串。字符串是程序将信息展示给人的方式。你可以打印它们，可以将它们写入文件，还可以将它们发送给网站服务器，很多事情都是通过字符串交流实现的。

字符串是非常好用的东西，所以在这节练习中你将学会如何创建包含变量内容的字符串。使用专门的格式和语法把变量的内容放到字符串里，相当于来告诉 `python` ：“嘿，这是一个格式化字符串，把这些变量放到那几个位置。”

一样的，即使你读不懂这些内容，只要一字不差地输入就可以了。

```
my_name = 'Zed A. Shaw'
my_age = 35 # not a lie
my_height = 74 # inches
my_weight = 180 # lbs
my_eyes = 'Blue'
my_teeth = 'White'
my_hair = 'Brown'

print "Let's talk about %s." % my_name
print "He's %d inches tall." % my_height
print "He's %d pounds heavy." % my_weight
print "Actually that's not too heavy."
print "He's got %s eyes and %s hair." % (my_eyes, my_hair)
print "His teeth are usually %s depending on the coffee." % my_teeth

# this line is tricky, try to get it exactly right
print "If I add %d, %d, and %d I get %d." % (
    my_age, my_height, my_weight, my_age + my_height + my_weight)
```

> **Warning:**如果你使用了非 ASCII 字符而且碰到了编码错误，记得在最顶端加一行 `# -- coding: utf-8 --` 。

你应该看到的结果

```
$ python ex5.py
Let's talk about Zed A. Shaw.
He's 74 inches tall.
He's 180 pounds heavy.
Actually that's not too heavy.
He's got Blue eyes and Brown hair.
His teeth are usually White depending on the coffee.
If I add 35, 74, and 180 I get 289.
```

附加题

> 1.修改所有的变量名字，把它们前面的`my_`去掉。确认将每一个地方的都改掉，不只是你使用`=`赋值过的地方。2.试着使用变量将英寸和磅对应转换成厘米和千克。不要直接键入答案。使用 Python 的计算功能来完成。3.在网上搜索所有的 Python 格式化字符。4.试着使用更多的格式化字符。例如`%r`就是是非常有用的一个，它的含义是“不管什么都打印出来”。

常见问题

Q:我可以定义一个类似`1 = 'Zed Shaw'`的变量吗?

> 不可以，`1`不是一个合法的变量名.变量需要以字母开头 ，比如`a1`才是正确的变量命名。

Q:这些字符`%s`，`%r`，`%d`是做什么的?

> 它们都是“格式化字符串”，你继续学习下去，就会学到关于它们更多的知识。它们告诉 Python 用后面的变量值代替字符串中的符号`%s`。那么什么是“格式化字符串呢”？ 我也说不清楚。教你学会编程有一个难题就是想要理解我在说什么，你必须先学会怎样编程。解决这个难题的办法就是先按照我的要求做我让你做的事情，后面会慢慢解释。如果你遇到一些类似的问题，你可以先记下来，后面我会慢慢解答。

Q:怎样生成一个浮点数?

> 你可以像这样`round(1.7333)`使用函数`round()`。

Q:我遇到在这个报错信息:`'str' object is not callable.`

> 你可能忘记在字符串以及变量之间输入`%`。

Q:为什么这个练习对我没有意义?

> 用你自己的数据修改脚本中的数字，看起来挺奇怪的，但是这些真实的信息能让这个练习更加真实，而且，你才刚刚开始学习，确实也不会有太大的意义，坚持做更多的练习题，你会有所收获。

练习6.字符串和文本

虽然你已经在程序中写过字符串了，你还没学过它们的用处。在这节练习中我们将使用复杂的字符串来建立一系列的变量，从中你将学到它们的用途。首先我们解释一下字符串是什么。

字符串通常是指你想要展示给别人的、或者是你想要从程序里“导出”的一小段字符。Python 可以通过文本里的双引号 `"` 或者单引号 `'` 识别出字符串来。这在你以前的 print 练习中你已经见过很多次了。如果你把单引号或者双引号括起来的文本放到 print 后面，它们就会被 python 打印出来。

字符串可以包含格式化字符 `%s` ，这个你之前也见过的。你只要将格式化的变量放到字符串中，再紧跟着一个百分号 `%` (percent)，再紧跟着变量名即可。 唯一要注意的地方，是如果你想要在字符串中通过格式化字符放入多个变量的时候，你需要将变量放到 `( )` 圆括号(parenthesis)中，而且变量之间用 `,` 逗号(comma)隔开。就像你逛商店说“我要买牛奶、面包、鸡蛋、汤”一样，只不过程序员说的是”(milk, eggs, bread, soup)”。

我们将练习输入大量的字符串、变量、和格式化字符，并且将它们打印出来。我们还将练习使用简写的变量名。程序员喜欢用高难度的简写来节约打字时间，所以我们现在就提早学会这个，这样你就能读懂并且写出这些东西了。

```
x = "There are %d types of people." % 10
binary = "binary"
do_not = "don't"
y = "Those who know %s and those who %s." % (binary, do_not)

print x
print y

print "I said: %r." % x
print "I also said: '%s'." % y

hilarious = False
joke_evaluation = "Isn't that joke so funny?! %r"

print joke_evaluation % hilarious

w = "This is the left side of..."
e = "a string with a right side."

print w + e
```

你看到的结果

```
$ python ex6.py
There are 10 types of people.
Those who know binary and those who don't.
I said: 'There are 10 types of people.'.
I also said: 'Those who know binary and those who don't.'.
Isn't that joke so funny?! False
This is the left side of...a string with a right side.
```

附加题

> 1. 通读程序，并给每一行加上注释，解释下这行的作用。
> 2. 找到所有的"字符串包含字符串"的位置，总共有四个位置。
> 3. 你确定只有四个位置吗？你怎么知道的？也许我在骗你呢。
> 4. 解释一下为什么用 `+` 连起来 w 和 e 就可以生成一个更长的字符串。

常见问题

Q: `%r` 和 `%s` 有什么不同?

> 用 `%r` 显示的是变量"原始"的数据值， `%r` 在打印的时候能够重现它代表的对象，但其他的符号用来给用户显示变量值。看下面的例子理解一下：
>
> > text = "I am %d years old." % 22 > > print "I said: %s." % text > > print "I said: %r." % text
>
> 返回的结果：
>
> > I said: I am 22 years old.. > > I said: 'I am 22 years old.'. // %r 给字符串加了单引号

Q:我遇到这个报错:not all arguments converted during string formatting.

> 你要重新检查你的代码是否跟示例中的一样。发生这个错误的原因是你写的 `%` 的格式化字符串数量大于你给出的变量数量。再检查一遍，看你的代码哪里出错了。

Q:为什么用 `'(单引号)` 标识字符串而不是其他的符号?

> 大部分情况下这只是一种风格，在一个用双引号标识的字符串内部我也会用单引号来标识其中子串。看看代码的第10行我是如何使用单双引号的。如果你认为一个笑话很好笑，你能否些l `hilarious = True` ? 答案当时是可以，而且，我们会在习题27中学到布尔值 。

练习7.更多的打印（输出）

现在我们将做一批练习，在练习的过程中你需要输入代码，并让它们运行起来。我不会解释太多，因为这节的内容都是以前熟悉过的。这节练习的目的是巩固你学到的东西。我们几个练习后再见。不要跳过这些习题。不要复制粘贴！

```
print "Mary had a little lamb."
print "Its fleece was white as %s." % 'snow'
print "And everywhere that Mary went."
print "." * 10  # what'd that do?

end1 = "C"
end2 = "h"
end3 = "e"
end4 = "e"
end5 = "s"
end6 = "e"
end7 = "B"
end8 = "u"
end9 = "r"
end10 = "g"
end11 = "e"
end12 = "r"

# watch that comma at the end.  try removing it to see what happens
print end1 + end2 + end3 + end4 + end5 + end6,
print end7 + end8 + end9 + end10 + end11 + end12
```

你看到的结果

```
$ python ex7.py
Mary had a little lamb.
Its fleece was white as snow.
And everywhere that Mary went.
..........
Cheese Burger
```

附加题

> 1.逆向阅读，给每一行的加上注释。2.倒着朗读出来，找出自己的错误。3.从现在开始，把你犯过的错误记录一张纸上。4.在开始下一节习题时，阅读一遍你记录下来的错误，并且尽量避免在下个练习中再犯同样的错误。5.记住，每个人都会犯错误。程序员和魔术师一样，他们希望大家认为他们从不犯错，不过这只是表象而已，他们每时每刻都在犯错。

常见问题

Q: 为什么使用名字为'snow'的变量?

> 这个可不是一个变量，这只是一个字符串，变量的两边可不会出现单引号。

Q:有必要像你在附加题1中说的那样，给每一行代码加上英文注释吗？

> 也不是，你给每一行加上注释，只是方便你理解每一行代码的功能，不过，有时候当你要编码解决一个较难的问题时，还是需要加上注释的，这样能训练你将代码翻译成自己的语言。

Q:我可以用单引号或双引号标识一个字符串，那它们有什么不同吗?

> 在Python中，单双引号都可以用来标识一个字符串，单引号更多用在较短的字符串上。

Q:能不能不用逗号，而把最后两行合并到一行的 `print` 里?

> 当然可以，你能很容易做到这一点，但是这一行会变的很长，会超过80个字符，这在Python中可不是好的代码风格。

练习8.打印, 打印

现在，我们学习一个字符串怎样使用更复杂的格式化字符。下面的代码看起来很复杂，但是如果你能将代码分段，并给每一行加上注释，你一样可以弄明白代码的意思。

```
formatter = "%r %r %r %r"

print formatter % (1, 2, 3, 4)
print formatter % ("one", "two", "three", "four")
print formatter % (True, False, False, True)
print formatter % (formatter, formatter, formatter, formatter)
print formatter % (
    "I had this thing.",
    "That you could type up right.",
    "But it didn't sing.",
    "So I said goodnight."
)
```

你看到结果

```
$ python ex8.py
1 2 3 4
'one' 'two' 'three' 'four'
True False False True
'%r %r %r %r' '%r %r %r %r' '%r %r %r %r' '%r %r %r %r'
'I had this thing.' 'That you could type up right.' "But it didn't sing." 'So I said g
oodnight.'
```

仔细研究一下，并尝试分析下我是怎样在一个格式化字符串内部嵌套使用格式化的

附加题

> 1.自己检查结果，记下你犯过的错误，并且在下个练习中尽量不犯同样的错误。2.注意最后一行程序中既有单引号又有双引号，你觉得它是如何工作的？

常见问题

Q:我可以用 `%s` 或者 `%r` 来格式化字符串吗？

> 可以用 `%s` ，但是尽量在做调试的时候使用 `%r` 。%r显示的是变量“原始”的数据值，%r在打印的时候能够重现它代表的对象。

Q:为什么给 `one` 使用引号，而不给 `True` 和 `False` 使用？

> Python识别 `True` 和 `False` 表示真假的概念。如果你给它们加上引号，它们就变成了字符串，而不能用来判别真假了。在后面的习题27里，你会学到这些布尔值是如何运行的。

Q: 我尝试在字符串中输入一些中文(或者其它非ASCII字符)，但是 `%r` 打印出一些奇怪的符号。

> 换成 `%s` 试试，就能正常打印啦。

Q:为什么有时候我写的是双引号，而 `%r` 打印输出的是单引号？

> Python可以用最有效的方式打印输出字符串，而不是直接复制你写的代码。你说的情况是很正常的，因为 `%r` 常用来调试或检查，因此没必要将它输出的很漂亮。

Q:这些代码在Python3中为什么没有执行?

> 不要用Python3.用python2.7会好一些，最好用Python2.6。

Q:我可以用IDE来执行程序吗？

> 不行，你现在需要学习使用命令行模式。这是你开始学习编程最好的方法，对你学习编程是很重要的。IDE不利于你使用本书学习编程。

练习9.打印, 打印, 打印

现在，你应该明白这本书的模式是用做很多的练习来教会你新知识。我以你可能不明白的代码开始教学，然后在用更多的习题去解释一些概念。即使你现在并不明白，经过大量的练习之后你也能明白了。记录你不明白的地方，然后坚持做练习题，你会收获更多。

```
# Here's some new strange stuff, remember type it exactly.

days = "Mon Tue Wed Thu Fri Sat Sun"
months = "Jan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"

print "Here are the days: ", days
print "Here are the months: ", months

print """
There's something going on here.
With the three double-quotes.
We'll be able to type as much as we like.
Even 4 lines if we want, or 5, or 6.
"""
```

你看到的结果

```
$ python ex9.py
Here are the days:  Mon Tue Wed Thu Fri Sat Sun
Here are the months:  Jan
Feb
Mar
Apr
May
Jun
Jul
Aug

There's something going on here.
With the three double-quotes.
We'll be able to type as much as we like.
Even 4 lines if we want, or 5, or 6.
```

附加题

> 1. 自己检查结果，记录你犯过的错误，并且在下个练习中尽量不犯同样的错误。

常见问题

Q: 为什么当我使用 `\n` 来换行的时候， `%r` 就不生效了？

> 这就是 `%r` 格式的工作原理；你如何输入的，它就如何打印输出。

Q:为什么当我在3个双引号中间输入空格的时候，会报错？

> 你应该这样写`"""`而不是`" " "`,意思是说每两个双引号之间不能有空格。

Q:我的错误经常是一些拼写错误，这样是不是很差?

> 在编程初期，大部分错误都是一些简单的错误，比如拼写问题，错别字，或者是将简单的事情搞砸。这很正常，不用担心，但是也要尽量不犯相同的错误。

练习10.那是什么?

在习题 9 中我们接触了一些新东西。我让你看到两种让字符串扩展到多行的方法。第一种方法是在月份之间用 `\n` (back-slash n)隔开。 这两个字符的作用是在该位置上放入一个“新行(new line)”字符。

使用反斜杠 `\`(back-slash) 可以将难打印出来的字符放到字符串。针对不同的符号有很多这样的所谓“转义序列(escape sequences)”，但有一个特殊的转义序列，就是 双反斜杠(double back-slash) `\\` 。这两个字符组合会打印出一个反斜杠来。接下来我们做几个练习，然后你就知道这些转义序列的意义了。

另外一种重要的转义序列是用来将单引号 ' 和双引号 " 转义。想象你有一个用双引号引用起来的字符串，你想要在字符串的内容里再添加一组双引号进去，比如你想说 `"I "understand" joe."` ，Python 就会认为 "understand" 前后的两个引号是字符串的边界，从而把字符串弄错。你需要一种方法告诉 python 字符串里边的双引号是字符串而不是真正的双引号。

要解决这个问题，你需要将双引号和单引号转义，让 Python 将引号也包含到字符串里边去。这里有一个例子：

```
"I am 6'2\" tall."  # 将字符串中的双引号转义
'I am 6\'2" tall.'  # 将字符串中的单引号转义
```

第二种方法是使用“三引号(triple-quotes)”，也就是 `"""` ，你可以在一组三引号之间放入任意多行的文字。接下来你将看到用法。

```
tabby_cat = "\tI'm tabbed in."
persian_cat = "I'm split\non a line."
backslash_cat = "I'm \\ a \\ cat."

fat_cat = """
I'll do a list:
\t* Cat food
\t* Fishies
\t* Catnip\n\t* Grass
"""

print tabby_cat
print persian_cat
print backslash_cat
print fat_cat
```

你看到的结果

注意你打印出来的制表符，这节练习中的文字间隔对于答案的正确性是很重要的。

```
$ python ex10.py
        I'm tabbed in.
I'm split
on a line.
I'm \ a \ cat.

I'll do a list:
        * Cat food
        * Fishies
        * Catnip
        * Grass
```

转义序列

转义字符	实现功能
\	Backslash ()
\'	Single-quote (')
\"	Double-quote (")
\a	ASCII bell (BEL)
\b	ASCII backspace (BS)
\f	ASCII formfeed (FF)
\n	ASCII linefeed (LF)
\N{name}	Character named name in the Unicode database (Unicode only)
\r ASCII	Carriage Return (CR)
\t ASCII	Horizontal Tab (TAB)
\uxxxx	Character with 16-bit hex value xxxx (Unicode only)
\Uxxxxxxxx	Character with 32-bit hex value xxxxxxxx (Unicode only)
\v	ASCII vertical tab (VT)
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

这里有一小段有意思的代码，尝试说明它们实现了什么功能：

```
while True:
    for i in ["/","-","|","\\","|"]:
        print "%s\r" % i,
```

附加题

> 1. 通过把它们写在卡片上记住所有的转义序列。
> 2. 使用 `'''` (三个单引号)取代三个双引号，看看效果是不是一样的？
> 3. 结合转义序列和格式字符串创建一个更复杂的格式。
> 4. 记得 %r 格式化字符串吗？使用 %r 搭配单引号和双引号转义字符打印一些字符串出来。 将 %r 和 %s 比较一下。 注意到了吗？%r 打印出来的是你写在脚本里的内容，而 %s 打印的是你应该看到的内容。

常见问题

Q:如果我想把所有的月份写在新的一行上，应该怎么做?

> 像这样写就可以: `"\nJan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"`

Q: 我还没有完全弄明白最后一句代码，我应该继续研究吗？

> 当然要继续。把每次练习题中你不明白的地方记下来。当你完成更多的练习的时候，定期检查你的笔记，看看你是否可以明白笔记中的内容。有时候你可能需要回去看看之前做过的练习，并且重复的完成它们。

Q: 是什么让 `\\` 不同于其他的转义字符?

> 这是一种简单的写出 (`\`)字符的方法. 自己想想为什么我们需要 `\\`

Q:为什么我写 `//` 或者 `/n` 的时候，代码没有生效。

> 因为你用的是 `/` 而不是 `\`.这两个是不同的字符串，他们的作用也是不一样的。

Q:当我使用 `%r` 格式的时候，转义字符都没有生效。

> 因为 `%r` 打印出来的是你写在脚本里的内容, 这当然也会包含原始的转移序列的字符。可以使用 `%s` 。一定要记住： `%r` 是调试用的，而 `%s` 才是显示输出用的。

Q:我没有明白附加题3.你所说的“结合”转义序列和格式是什么意思？

> 你需要明白一点，所有的这些练习题，都可以结合起来解决一些难题。这节练习带你了解了格式化字符串，你可以结合使用格式化字符串和转义字符写一些新的代码。

Q: `'''` 和 `"""` 哪个更好?

> 这个只依赖于你的代码风格。 现在可以使用 `'''` (三个单引号)，但是也要做好准备别人都在用的，感觉更好的方式。

练习11.提问

从这节开始我们要恢复之前的脚步。我已经出过很多打印相关的练习，让你习惯写简单的东西，但简单的东西都有点无聊。我们现在要做的是把数据读到你的程序里边去。这可能对你有点难度，你可能一下子不明白，不过你需要相信我，无论如何把习题做了再说。只要做几个练习你就明白了。

一般软件做的事情主要就是下面几条：

> 1. 接收人的输入.
> 2. 改变输入.
> 3. 打印改变后的输入值

到目前为止你只做了打印字符串，但还不会接收或者修改人的输入。你也许还不知道“输入(input)”是什么意思。但是在代码中输入这个单词还是跟以前一样的，所以闲话少说，我们还是开始做点练习看你能不能明白。下一个习题里边我们会给你更多的解释。

```
print "How old are you?",
age = raw_input()
print "How tall are you?",
height = raw_input()
print "How much do you weigh?",
weight = raw_input()

print "So, you're %r old, %r tall and %r heavy." % (
    age, height, weight)
```

> **NOTE:** 注意我在每行 print 后面加了个逗号(comma) `,` 了吧？这样的话 print 就不会输出新行符而结束这一行跑到下一行去了。

你看到的结果

```
$ python ex11.py
How old are you? 38
How tall are you? 6'2"
How much do you weigh? 180lbs
So, you're '38' old, '6\'2"' tall and '180lbs' heavy.
```

附加题

> 1. 上网查一下 Python 的 `raw_input` 实现的是什么功能。
> 2. 你能找到它的别的用法吗？测试一下你上网搜索到的例子。
> 3. 用类似的格式再写一段，把问题改成你自己的问题
> 4. 结合转义序列，想想为什么最后一行 `'6\'2"'` 里边有一个 `\'` 序列。单引号需要被转义，从而防止它被识别为字符串的结尾。有没有注意到这一点？

常见问题

Q:如何接收用户输入的数字，用来进行数学计算？

> 这略微复杂一些,你可以试试用 `x = int(raw_input())` 将通过 `raw_input()` 获得的字符串转化成整数。

Q:我用 `raw_input("6'2")` 输入我的身高值，但是它没有生效。

> 你应该在你的终端里输入，而不是把输入值写到 `raw_input()` 的括号里。首先请检查你的代码是否和我提供的样例一样，然后执行这个脚本，当收到提示的时候，再输入你的身高。

Q:为什么你在第8行代码的时候换行了，而没有让这句代码在一行上？

> 这样做的目的是让一行代码少于80个字符，这是一种Python程序员喜欢的代码风格，如果你喜欢，你也可以把它们放在一行里。

Q: `input()` 和 `raw_input()` 有什么区别?

> 在Python代码里 `input()` 方法将会改变你输入的东西，但是这个方法存在安全问题，请尽量避免使用它。

Q:什么情况下我应该在输入的字符串前面加一个 `u` ，比如 `u'35'` ？

> 在Python中用这种方式告诉你这是一个Unicode编码的字符串。用 `%s` 格式可以让你正常打印。

练习12.提示别人

当你输入 `raw_input()` 的时候，你需要键入 `(` 和 `)` 也就是“括号(parenthesis)”。这和你格式化输出两个以上变量时的情况有点类似，比如说 "%s %s" % (x, y) 里边就有括号。对于 raw_input 而言，你还可以让它显示出一个提示，从而告诉别人应该输入什么东西。你可以在 () 之间放入一个你想要作为提示的字符串，如下所示：

```
y = raw_input("Name? ")
```

这句话会用 “Name?” 提示用户，然后将用户输入的结果赋值给变量 y。这就是我们提问用户并且得到答案的方式。

也就是说，我们的上一个练习可以使用 `raw_input` 重写一次。所有的提示都可以通过 `raw_input` 实现。

```
age = raw_input("How old are you? ")
height = raw_input("How tall are you? ")
weight = raw_input("How much do you weigh? ")

print "So, you're %r old, %r tall and %r heavy." % (
    age, height, weight)
```

你看到的结果

```
$ python ex12.py
How old are you?  38
How tall are you?  6'2"
How much do you weigh?  180lbs
So, you're '38' old, '6\'2"' tall and '180lbs' heavy.
```

附加题

> 1. 在命令行界面下运行你的程序，然后在命令行输入 `pydoc raw_input` 看它说了些什么。如果你用的是 Window，那就试一下 `python -m pydoc raw_input` 。
> 2. 输入 `q` 退出 `pydoc` 。
> 3. 上网找一下 `pydoc` 命令是用来做什么的。
> 4. 使用 `pydoc` 再看一下 `open` , `file` , `os` , 和 `sys` 的含义。看不懂没关系，只要通读一下，记下你觉得有意思的点就行了。

常见问题

Q:我运行 `pydoc` 的时候，为什么会遇到这个报错 `invalid syntax` ?

> 你没有在命令行里执行`pydoc`; 你是不是在启动`python`后执行的？退出Python试试吧.

Q:我执行 `pydoc` 的时候，我遇到一个提示 `pydoc 不是内部或外部命令` 。

> 有一些windows上的Python版本没有提供这个命令,你可以跳过这个附加练习，当你需要阅读Python 文档的时候，你在网上搜索就可以了。

Q:为什么用 `%r` 而不是 `%s` ?

> 请务必记住 `%r` 会原样输出你输入的每一个字符，而`%s`是用来显示你的输入的。下次，我不会再回答相同的问题。这是大家重复问到次数最多的问题，但是一遍一遍问相同的问题，说明你没有记住我讲过的内容。

Q:为什么不能这样输入 `"How old are you?" , raw_input()` ?

> 你觉得它会生效的, 但是Python认为这种写法是不合法的. 我能告诉你的也只能是你不能这样么写。

练习13.参数, 解包, 变量

在这节练习中，我们将学到另外一种将变量传递给脚本的方法(脚本就是你写的 `.py` 程序)。你已经知道，如果要运行 `ex13.py` ，只要在命令行运行 `python ex13.py` 就可以了。这句命令中的 `ex13.py` 部分就是所谓的“参数(argument)”，我们现在要做的就是写一个可以接收参数的脚本。

将下面的程序写下来，后面你会看到详细的解释：

```
from sys import argv

script, first, second, third = argv

print "The script is called:", script
print "Your first variable is:", first
print "Your second variable is:", second
print "Your third variable is:", third
```

第一行代码中，我们用到一个 `import` 语句，这是将Python的功能模块加入你自己脚本的方法。Python 不会一下子将它所有的功能提供给你，而是让你需要什么就调用什么。这样可以让你的程序更加精简，而后面的程序员看到你的代码的时候，这些“import”语句可以作为提示，让他们明白你的代码用到了哪些功能。

`argv` 就是所谓的“参数变量(argument variable)”，它是一个非常标准的编程术语。在其他的编程语言里你也可以看到它。这个变量包含了你传递给 Python 的参数。通过后面的练习你将对它有更多的了解。

代码的第3行将 `argv` 进行“解包(unpack)”，与其将所有参数放到同一个变量下面，我们将每个参数赋予一个变量名： script, first, second, 以及 third。这也许看上去有些奇怪,不过”解包”可能是最好的描述方式了。它的含义很简单：“把 argv 中的东西解包，将所有的参数依次赋予左边的变量名”。

接下来，就是正常的打印输出了。

等一下! 功能还有另外一个名字

前面我们使用 `import` 让你的程序实现更多的功能，把 `import` 称为“功能”。我希望你可以在没接触到正式术语的时候就弄懂它的功能。在继续下去之前, 你需要知道它们的真正名称：模块(modules)。

从现在开始我们将把这些我们导入(import)进来的功能称作模块。你将看到类似这样的说法：“你需要把sys模块import进来。”也有人将它们称作“库(libraries)”，不过我们还是叫它们模块吧。

你看到的结果

像下面的示例一样将你的脚本运行起来（你必须在命令行里传递3个参数）：

```
$ python ex13.py first 2nd 3rd
The script is called: ex13.py
Your first variable is: first
Your second variable is: 2nd
Your third variable is: 3rd
```

如果你每次输入的参数不一样，那你看到的输出结果也会略有不同：

```
$ python ex13.py stuff things that
The script is called: ex13.py
Your first variable is: stuff
Your second variable is: things
Your third variable is: that
$
$ python ex13.py apple orange grapefruit
The script is called: ex13.py
Your first variable is: apple
Your second variable is: orange
Your third variable is: grapefruit
```

你可以将 `first` , `2nd` , 和 `3rd` 替换成任意你喜欢的3个参数

如果你没有运行对，你可能会看到的错误信息：

```
$ python ex13.py first 2nd
Traceback (most recent call last):
  File "ex13.py", line 3, in <module>
    script, first, second, third = argv
ValueError: need more than 3 values to unpack
```

附加题

> 1. 给你的脚本三个以下的参数。看看会得到什么错误信息。试着解释一下。
> 2. 再写两个脚本，其中一个接受更少的参数，另一个接受更多的参数，在参数解包时给它们取一些有意义的变量名。
> 3. 将 `raw_input` 和 `argv` 一起使用，让你的脚本从用户手上得到更多的输入。
> 4. 记住“模块(modules)”为你提供额外功能。多读几遍把这个词记住，因为我们后面还会用到它。

常见问题

Q: 当我运行脚本的时候，有个报错: `need more than 1 value to unpack` .

> 一定记住，关注细节是学习编程的三要素之一。如果你仔细看了我是如何在命令行运行脚本的，你也能把你的脚本正确的运行起来。

Q: `argv` 和 `raw_input()` 有什么区别?

> 它们的不同之处在于要求用户输入的位置不同。如果你想让用户在命令行输入你的参数，你应该使用`argv`.，如果你希望用户在脚本执行的过程中输入参数，那就就要用到`raw_input()`。

Q:命令行输入的参数是字符串吗?

> 是的，如果你需要输入数字，可以使用`int()`把他们转化成整数，可以参考`int(raw_input())`.

Q:如何使用命令行?

> 通过这节练习，你其实已经快速的学会如何使用命令行了，如果在此阶段你想深入学习的话，你可以阅读这本书的附录A--命令行速成教程。

Q: 我不知道怎样把`argv` 和 `raw_input()`结合起来使用.

> 不要想太多. 用`raw_input()`修改脚本后面的两行代码，然后打印输出就行。然后试试用更多的方式来同时使用这两种方法修改脚本。

Q:为什么我不能这么写 `raw_input('? ') = x` ?

> 因为这种写法正好是将它正常运行方法的逆序，按照我的写法修改，就能正常运行了。

练习14.提示和传递

这节练习让我们使用 `argv` 和 `raw_input` 一起来向用户提一些特别的问题。下一节习题你会学习如何读写文件，这节练习是下节的基础。在这道习题里我们将用略微不同的方法使用 `raw_input` ，让它打出一个简单的 `>` 作为提示符。这和一些游戏中的方式类似，例如 Zork 或者 Adventure 这两款游戏。

```
from sys import argv

script, user_name = argv
prompt = '> '

print "Hi %s, I'm the %s script." % (user_name, script)
print "I'd like to ask you a few questions."
print "Do you like me %s?" % user_name
likes = raw_input(prompt)

print "Where do you live %s?" % user_name
lives = raw_input(prompt)

print "What kind of computer do you have?"
computer = raw_input(prompt)

print """
Alright, so you said %r about liking me.
You live in %r.  Not sure where that is.
And you have a %r computer.  Nice.
""" % (likes, lives, computer)
```

我们将用户提示符设置为变量 `prompt` ，这样我们就不需要在每次用到 `raw_input` 时重复输入提示用户的字符了。而且如果你要将提示符修改成别的字串，你只要改一个位置就可以了。

是不是非常方便？

你看到的结果

当你运行这个脚本时，记住你需要把你的名字赋给这个脚本，让 argv 参数接收到你的名称。

```
$ python ex14.py zed
Hi zed, I'm the ex14.py script.
I'd like to ask you a few questions.
Do you like me zed?
>  Yes
Where do you live zed?
>  San Francisco
What kind of computer do you have?
>  Tandy 1000

Alright, so you said 'Yes' about liking me.
You live in 'San Francisco'.  Not sure where that is.
And you have a 'Tandy 1000' computer.  Nice.
```

附加题

> 1. 查一下 Zork 和 Adventure是两个怎样的游戏。看看能不能下载到一版，然后玩玩看。
> 2. 将 `prompt` 变量改成完全不同的内容再运行一遍。
> 3. 给你的脚本再添加一个参数，让你的程序用到这个参数，像你在上一个练习中用到的解包操作 `first, second = ARGV` 。
> 4. 确认你弄懂了三个引号 `"""` 可以定义多行字符串，而 `%` 是字符串的格式化工具。

常见问题

Q:我遇到一个报错 `SyntaxError:invalid syntax` .

> 重申一遍，你要在命令行里执行脚本，而不是在Python的解释器里。如果在python解析器里再输入 `python ex14.py Zed` 一定会出错的，因为你是在Python里运行Python脚本了。

Q:我不太明白你说的改变变量 `prompt` 的值?

> 看到代码的这句了么 `prompt = '> '` ，改变它的值就是让你修改 `prompt` 为一个不同于 `>` 的值，只是修改一个字符串而已，你已经做过13个练习题，这次试试看自己解决这个问题。

Q:我遇到一个错误信息 `need more than 1 value to unpack` .

> 看看“你看到的结果”这部分，并且看看我是怎么运行脚本的。你应该注意我是如何输入命令的，为什么我输入了一个命令行参数？

Q:怎样在IDLE运行程序?

> 不要用IDLE，现在只用文本编辑器就够了。

Q:我能给变量 `prompt` 用双引号吗？

> 当然可以，尽管试试看。

Q:我遇到一个明明错误: `name 'prompt' is not defined` .

> 你可能是拼写错了prompt或者是丢掉了代码的那一行。逐行对比下你和我的代码那里不一样。你遇到这类错误意味着你有拼写错误或者你根本没有定义这个变量。

练习15.读文件

你已经学过 `raw_input` 和 `argv` ，这些是你开始学习读取文件的必备基础。你可能需要多多实验才能明白它的工作原理，所以你要细心做练习，并且仔细检查结果。处理文件需要非常仔细，否则，你可能会把有用的文件弄坏或者清空。导致前功尽弃。

这节练习涉及到写两个文件。一个正常的 ex15.py 文件，另外一个是 ex15_sample.txt，第二个文件并不是脚本，而是供你的脚本读取的文本文件。以下是后者的内容：

```
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.
```

我们要做的是用我们的脚本“打开(open)”这个文件，然后打印出来。然而把文件名 `ex15_sample.txt` 写死在代码中并不是一个好主意，这些信息应该是用户输入的才对。如果我们碰到其他文件要处理，写死的文件名就会给你带来麻烦了。我们的解决方案是使用 `argv` 和 `raw_input` 来从用户获取信息，从而知道哪些文件该被处理。

```
from sys import argv

script, filename = argv

txt = open(filename)

print "Here's your file %r:" % filename
print txt.read()

print "Type the filename again:"
file_again = raw_input("> ")

txt_again = open(file_again)

print txt_again.read()
```

这个脚本中有一些新奇的玩意，我们来快速地过一遍：

代码的 1-3 行使用 `argv` 来获取文件名，这个你应该已经熟悉了。接下来第 5 行我们看到 `open` 这个新命令。现在请在命令行运行 `pydoc open` 来读读它的说明。你可以看到它和你自己的脚本、或者 `raw_input` 命令类似，它会接受一个参数，并且返回一个值，你可以将这个值赋予一个变量。这就是你打开文件的过程。

第 7 行我们打印了一小行信息，但在第 8 行我们看到了新奇的东西。我们在 `txt` 上调用了一个函数。你从 `open` 获得了一个文件，文件本身也支持一些命令。它接受命令的方式是使用句点 `.` (英文称作 dot 或者 period)，紧跟着你的命令，然后是类似 `open` 和 `raw_input` 一样的参数。不同点是：当你执行 `txt.read` 时，你的意思其实是：“嘿 txt！执行你的 read 命令，无需任何参数！”

脚本剩下的部分基本差不多，不过我就把剩下的分析作为附加题留给你自己了。

你看到的结果

我创建了一个名字叫做 `ex15_sample.txt` 的文件，然后执行我的脚本：

```
$ python ex15.py ex15_sample.txt
Here's your file 'ex15_sample.txt':
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.

Type the filename again:
>  ex15_sample.txt
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.
```

附加题

这节的难度跨越有点大，所以你要尽量做好这节加分习题，然后再继续后面的章节。

> 1. 在每一行的上面加上注释。
> 2. 如果你不确定答案，就问别人，或者上网搜索。大部分时候，只要搜索 “python” 加上你要搜的东西就能得到你要的答案。比如搜索一下“python open”。
> 3. 我使用了“命令”这个词，不过实际上它们的名字是“函数（function）”和“方法（method）。上网搜索一下这两者的意义和区别。看不明白也没关系，这本书后面也会讲到这些。
> 4. 删掉 10-15 行使用到 `raw_input` 的部分，再运行一遍脚本。
> 5. 只用 `raw_input` 写这个脚本，想想哪种得到文件名称的方法更好？为什么？
> 6. 运行 python 在命令行下使用 open 打开一个文件，这种 open 和 read 的方法也值得你一学。
> 7. 让你的脚本对 `txt` 和 `txt_again` 两个变量执行一下 `close()` ，处理完文件后你需要将其关闭，这是很重要的一点。

常见问题

Q: `txt = open(filename)` 返回的是文件的文本内容吗？

> 不是的。它的返回值我们称为“文件对象”。你可以把文件想象成19世纪50年代的大型计算机上的老旧的磁带驱动器， 或者是像现在的DVD播放器，你可以在他们内部走动，然后阅读他们。但是文件对象并不是文件的文本内容一样就好像DVD播放器也不是一个DVD视频.

Q:我不能按照你在附加题7中说的那样在命令行输入代码

> 首先，在命令行里输入`python`并回车，现在你已经进入了一个python解析器。接下来你就可以输入一系列的代码，python会一一执行你的代码。最后别忘了输入`quit()`并回车退出python。

Q:当我打开同一个文件两次的时候，为什么不会报错？

> Python不会限制你只能打开一个文件一次,有时这是必要的。

Q: `from sys import argv` 这句是什么意思?

> 目前来说，你可以认为`sys`是一个包，这句代码的意思是从`sys`的包中引入`argv`功能模块。

Q: 我把文件的名字放进脚本中，`ex15_sample.txt = argv`，却没有生效。

> 你不能这么写，请按照我的示例写代码，并像我一样在命令行里运行脚本。

练习16.读写文件

如果你做了上一个练习的附加题部分，你应该已经了解了文件相关的各种命令（方法/函数）。下面的这张表，是你应该记住的命令：

> - close -- 关闭文件。跟你编辑器的 文件->保存.. 一个意思。
> - read -- 读取文件内容。你可以把结果赋给一个变量。
> - readline -- 读取文本文件中的一行。
> - truncate -- 清空文件，请谨慎使用该命令。
> - write('stuff') -- 将stuff写入文件。

这是你现在该知道的重要命令。有些命令需要接受参数，这对我们并不重要。你只要记住 `write` 需要接收一个字符串作为参数，从而将该字符串写入文件。

让我们来使用这些命令做一个简单的文本编辑器吧:

```
from sys import argv

script, filename = argv

print "We're going to erase %r." % filename
print "If you don't want that, hit CTRL-C (^C)."
print "If you do want that, hit RETURN."

raw_input("?")

print "Opening the file..."
target = open(filename, 'w')

print "Truncating the file.  Goodbye!"
target.truncate()

print "Now I'm going to ask you for three lines."

line1 = raw_input("line 1: ")
line2 = raw_input("line 2: ")
line3 = raw_input("line 3: ")

print "I'm going to write these to the file."

target.write(line1)
target.write("\n")
target.write(line2)
target.write("\n")
target.write(line3)
target.write("\n")

print "And finally, we close it."
target.close()
```

这个文件是够大的，大概是你创建过的最大的文件。所以慢慢来，仔细检查，让它能运行起来。有一个小技巧就是你可以让你的脚本一部分一部分地运行起来。先写 1-8 行，让它运行起来，再多运行5行，再接着多运行几行，以此类推，直到整个脚本运行起来为止。

你看到的结果

你将看到两样东西，一个是你脚本的输出:

```
$ python ex16.py test.txt
We're going to erase 'test.txt'.
If you don't want that, hit CTRL-C (^C).
If you do want that, hit RETURN.
?
Opening the file...
Truncating the file.  Goodbye!
Now I'm going to ask you for three lines.
line 1:  Mary had a little lamb
line 2:  Its fleece was white as snow
line 3:  It was also tasty
I'm going to write these to the file.
And finally, we close it.
```

接下来打开你新建的文件（我的是test.txt）检查一下里边的内容，怎么样，不错吧？

附加题

> 1. 如果你觉得自己没有弄懂的话，用我们的老办法，在每一行之前加上注解，为自己理清思路。就算不能理清思路，你也可以知道自己究竟具体哪里没弄明白。
> 2. 写一个和上一个练习类似的脚本，使用 `read` 和 `argv` 读取你刚才新建的文件。
> 3. 文件中重复的地方太多了。试着用一个 `target.write()` 将 line1, line2, line3 打印出来，你可以使用字符串、格式化字符、以及转义字符。
> 4. 找出为什么我们需要给 open 多赋予一个 'w' 参数。提示： open 对于文件的写入操作态度是安全第一，所以你只有特别指定以后，它才会进行写入操作。
> 5. 如果你使用'w' 模式打开一个文件，那么还需要 `target.truncate()` 吗?阅读Python关于 `open` 函数的文档， 看你理解的对不对。

常见问题

Q: `truncate()` 方法必须要有参数'w'吗?

> 参考附加题5

Q: 'w'参数是什么意思?

> 它只是打开文件的一种模式。如果你用了这个参数，表示"以写（ `write` ）模式打开文件。同样有 `'r'` 表示只读模式， `'a'` 表示追加模式，还有一些其他的修饰符。

Q: 有什么修饰符我可以用在打开文件的模式上？

> 最重要的一个就是`+`, 使用它，你可以有`'w+'`, `'r+'`, 和`'a+'`模式. 这样可以同时以读写模式打开文件。

Q: `open(filename)`是以'r' (只读) 模式打开文件吗?

> 是的，'r' 是`open()`函数的默认参数值。

练习17.更多文件操作

现在让我们再学习几种文件操作。我们将编写一个Python脚本，将一个文件中的内容拷贝到另外一个文件中。这个脚本很短，不过它会让你对于文件操作有更多的了解。

```
from sys import argv
from os.path import exists

script, from_file, to_file = argv

print "Copying from %s to %s" % (from_file, to_file)

# we could do these two on one line, how?
in_file = open(from_file)
indata = in_file.read()

print "The input file is %d bytes long" % len(indata)

print "Does the output file exist? %r" % exists(to_file)
print "Ready, hit RETURN to continue, CTRL-C to abort."
raw_input()

out_file = open(to_file, 'w')
out_file.write(indata)

print "Alright, all done."

out_file.close()
in_file.close()
```

你应该很快注意到了我们 `import` 了又一个很好用的命令 `exists` 。这个命令将文件名字符串作为参数，如果文件存在的话，它将返回 `True` ，否则将返回 `False` 。在本书的下半部分，我们将使用这个函数做很多的事情，不过现在你应该学会怎样通过 import 调用它.

通过使用 `import` ，你可以在自己代码中直接使用其他更厉害的（通常是这样，不过也不 尽然）程序员写的大量免费代码，这样你就不需要重写一遍了。

你看到的结果

和你前面写的脚本一样，运行该脚本需要两个参数，一个是待拷贝的文件，一个是要拷贝至的文件。我再创建一个名为 `test.txt` 的测试文件，我们将看到如下的结果:

```
$ echo "This is a test file." > test.txt
```

$ cat test.txt This is a test file. $ $ python ex17.py test.txt new_file.txt Copying from test.txt to new_file.txt The input file is 21 bytes long Does the output file exist? False Ready, hit RETURN to continue, CTRL-C to abort.

```
Alright, all done.
```

该命令对于任何文件都应该是有效的。试试操作一些别的文件看看结果。不过小心别把你的重要文件给弄坏了。

> **Warning:**你看到我用 `echo` 命令创建一个文件，并用 `cat` 命令展示一个文件了吧？它们只能在 Linux 和 OSX 下使用，你可以阅读附录A获得者两个命令的使用方法。

附加题

> 1. 这个脚本 实在是有点烦人。没必要在拷贝之前问一遍把，没必要在屏幕上输出那么多东西。试着删掉脚本的一些功能，让它使用起来更加友好。
> 2. 看看你能把这个脚本改多短，我可以把它写成一行。
> 3. 我使用了一个叫 `cat` 的东西，这个古老的命令的用处是将两个文件“连接(con_cat_enate)”到一起，不过实际上它最大的用途是打印文件内容到屏幕上。你可以通过 `man cat` 命令了解到更多信息。(windows下没有这个命令)
> 4. 找出为什么你需要在代码中写 output.close() 。
> 5. 再多读读和 `import` 相关的材料，将python运行起来，试试这一条命令。试着看看自己能不能摸出点门道，当然了，即使弄不明白也没关系。

常见问题

Q: 为什么 `'w'` 要写在引号里？

> 它只是个字符串，你已经做过太多关于字符串的练习了，你知道什么是字符串的，对吗？

Q:总是感觉这些练习很难，这正常吗？

> 这很正常的。直到你做完练习36甚至完成这本书的学习，用python做出一些作品，编程对你来说可能都不是一件简单的事情。每个人都不一样，所以只要坚持复习你认为困难的习题，直到你真的搞明白它们,要有耐心。

Q:函数 `len()` 是干什么用的？

> 它能获得参数的长度，返回值是一个数字，你试着用用这个方法。

Q:我尝试改短代码的时候，在脚本的结尾处遇到一个关于文件关闭的问题。

> 你可能做了一些类似这样的事情，比如 `indata = open(from_file).read()`,这样写的话，就不需要在执行关闭操作，当执行完这一行的时候，文件自动就被关闭了。

Q: 我遇到一个异常 `Syntax:EOL while scanning string literal`

> 你应该是忘记在字符串的结尾加上引号了，回到你出错的那行代码，检查一下。

练习18.命名，变量，代码，函数

标题包含的内容够多的吧？接下来我要教你“函数(function)”了！说到函数，不一样的人会对它有不一样的理解和使用方法，不过我只会教你现在能用到的最简单的使用方式。

函数可以做三样事情：

> 1. 它们给代码片段命名，就跟“变量”给字符串和数字命名一样。
> 2. 它们可以接受参数，就跟你的脚本接受`argv`一样。
> 3. 通过使用 #1 和 #2，它们可以让你创建“微型脚本”或者“小命令”。

python中你可以使用`def`新建函数。我将让你创建四个不同的函数，它们工作起来和你的脚本一样。然后我会演示给你各个函数之间的关系。

```
# this one is like your scripts with argv
def print_two(*args):
    arg1, arg2 = args
    print "arg1: %r, arg2: %r" % (arg1, arg2)

# ok, that *args is actually pointless, we can just do this
def print_two_again(arg1, arg2):
    print "arg1: %r, arg2: %r" % (arg1, arg2)

# this just takes one argument
def print_one(arg1):
    print "arg1: %r" % arg1

# this one takes no arguments
def print_none():
    print "I got nothin'."

print_two("Zed","Shaw")
print_two_again("Zed","Shaw")
print_one("First!")
print_none()
```

让我们把地一个函数`print_two`分解一下，这个函数和你写脚本的方式差不多，因此你看上去应该会觉着比较眼熟：

> 1.首先我们告诉Python创建一个函数，我们使用到的命令是`def`，也就是“定义(define)”的意思。
>
> 1. 紧接着def的是函数的名称。本例中它的名称是`print_two`，但名字可以随便取，就叫`peanuts`也没关系。但函数名最好能够体现出函数的功能来。
> 2. 然后我们告诉函数我们需要`*args`，这和脚本的`argv`非常相似，参数必须放在圆括号 () 中才能正常工作。
> 3. 接着我们用冒号`:`结束本行，然后开始下一行缩进。
> 4. 冒号以下，使用4个空格缩进的行都是属于`print_two`这个函数的内容。 其中第一行的作用是将参数解包，这和脚本参数解包的原理差不多。
> 5. 为了演示它的工作原理，我们把解包后的每个参数都打印出来，这和我们在之前脚本练习中所作的类似。

函数 `print_two` 的问题是：它并不是创建函数最简单的方法。在 Python 函数中我们可以跳过整个参数解包的过程，直接使用`()`里边的名称作为变量名。这就是`print_two_again`实现的功能。

接下来的例子是`print_one`，它向你演示了函数如何接受单个参数。

最后一个例子是`print_none`，它向你演示了函数可以不接收任何参数。

> **Warning:**如果你不太能看懂上面的内容也别气馁。后面我们还有更多的练习向你展示如何创建和使用函数。现在你只要把函数理解成“迷你脚本”就可以了。

你看到的结果

运行上面的脚本会看到如下结果:

```
$ python ex18.py
arg1: 'Zed', arg2: 'Shaw'
arg1: 'Zed', arg2: 'Shaw'
arg1: 'First!'
I got nothin'.
$
```

你应该已经看出函数是怎样工作的了。注意到函数的用法和你以前见过的 `exists`、`open`，以及别的“命令”有点类似了吧？其实我只是为了让你容易理解才叫它们“命令”，它们的本质其实就是函数。也就是说，你也可以在自己的脚本中创建你自己的“命令”。

附加题

为自己写一个`函数注意事项`以供后续参考。你可以写在一个索引卡片上随时阅读，直到你记住所有的要点为止。注意事项如下：

> 1. 函数定义是以 `def` 开始的吗？
> 2. 函数名称是以字符和下划线 `_` 组成的吗？
> 3. 函数名称是不是紧跟着括号 `(` ？
> 4. 括号里是否包含参数？多个参数是否以逗号隔开？
> 5. 参数名称是否有重复？（不能使用重复的参数名）
> 6. 紧跟着参数的是不是括号和冒号 `):` ？
> 7. 紧跟着函数定义的代码是否使用了 4 个空格的缩进 (indent)？
> 8. 函数结束的位置是否取消了缩进 (“dedent”)？

当你运行（或者说“使用use”或“调用call”）一个函数时，记得检查下面的点：

> 1. 调运函数时是否使用了函数的名称？
> 2. 函数名称是否紧跟着 `(` ？
> 3. 括号后有无参数？多个参数是否以逗号隔开？
> 4. 函数是否以 `)` 结尾？

按照这两份检查表里的内容检查你的代码，直到你不需要检查表为止。

最后，将下面这句话阅读几遍：

```
‘运行函数(run)’、‘调用函数(call)’、和‘使用函数(use)’是同一个意思
```

常见问题

Q:什么字符允许用在函数名上？

> 和变量命名规则相同。不能以数字开头，并且只能包含字母数字和下划线。

Q: `*args` 中的星号 `*` 是干嘛的？

> 它告诉python把函数的所有参数组织一个列表放在 `args` 里。类似你之前用过的 `argv` ，只不过 `*args` 是用在函数里的，

Q: 这个练习让我觉得枯燥

> 这是好事，说明你在输入代码方面做的越来越好，而且也能很好的明白这些代码是做什么的，想让练习不那么枯燥，输入我让你练习的代码，然后故意出错，检查你的错误并修复它们。

练习19.函数和变量

函数这个概念也许承载了太多的信息量，不过别担心。只要坚持做这些练习，对照上个练习中的检查点检查一遍这次的联系，你最终会明白这些内容的。

有一个你可能没有注意到的细节，我们现在强调一下：函数里边的变量和脚本里边的变量之间是没有关系的。下面的这个练习可以让你对这一点有更多的思考：

```
def cheese_and_crackers(cheese_count, boxes_of_crackers):
    print "You have %d cheeses!" % cheese_count
    print "You have %d boxes of crackers!" % boxes_of_crackers
    print "Man that's enough for a party!"
    print "Get a blanket.\n"

print "We can just give the function numbers directly:"
cheese_and_crackers(20, 30)

print "OR, we can use variables from our script:"
amount_of_cheese = 10
amount_of_crackers = 50

cheese_and_crackers(amount_of_cheese, amount_of_crackers)

print "We can even do math inside too:"
cheese_and_crackers(10 + 20, 5 + 6)

print "And we can combine the two, variables and math:"
cheese_and_crackers(amount_of_cheese + 100, amount_of_crackers + 1000)
```

通过这个练习，你看到我们给函数 `cheese_and_crackers` 传递很多的参数，然后在函数里把它们打印出来。我们可以在函数里用变量名，可以在函数里做运算，甚至可以将变量和运算结合起来。

从一方面来说，函数的参数和我们的生成变量时用的 `=` 赋值符类似。事实上，如果你可以用 `=` 给一个东西命名，你也就可以将其作为参数传递给一个函数。

你看到的结果

你应该研究一下脚本的输出，和你想象的结果对比一下看有什么不同。

```
$ python ex19.py
We can just give the function numbers directly:
You have 20 cheeses!
You have 30 boxes of crackers!
Man that's enough for a party!
Get a blanket.

OR, we can use variables from our script:
You have 10 cheeses!
You have 50 boxes of crackers!
Man that's enough for a party!
Get a blanket.

We can even do math inside too:
You have 30 cheeses!
You have 11 boxes of crackers!
Man that's enough for a party!
Get a blanket.

And we can combine the two, variables and math:
You have 110 cheeses!
You have 1050 boxes of crackers!
Man that's enough for a party!
Get a blanket.
```

附加题

> 1. 倒着将脚本读完，在每一行上面添加一行注解，说明这行的作用。
> 2. 从最后一行开始，倒着阅读每一行，读出所有的重要字符来。
> 3. 自己编至少一个函数出来，然后用10种方法运行这个函数。

常见问题

Q: 怎么可能有10中方法来运行一个函数？

> 信不信由你,理论上有无数种方法去调用一个函数。看看你对函数、变量、用户输入有多少想象力和创造力。

Q:有没有一种方法分析一下这个函数是做什么的，这样能让我更方便的理解它？

> 有很多种方法可以做到,但是尽量采用给每行增加注释的方式。另一种方式是大声的读出代码。第三种方式是将代码打印在纸上，并添加图片和注释用来解释代码实现了什么功能。

Q: 如果我希望用户输入芝士和饼干的数量，我该怎么做？

> 你可以使用 `int()` 把你从 `raw_input()` 获取的参数转化为数字。

Q: 在函数中能否改变变量 `cheese_count` 和 `amount_of_cheese` 的值？

> 当然不能, 这些变量是独立的，在函数体之外。 他们被作为零食变量传递给函数是为了保证函数的正常运行，当函数退出的时候，这些临时变量也就消失了。继续学习本书，你会更明白这些。

Q: 定义一个和函数名相同名字的全局变量是不是不好？

> 当然不好, 如果你这么做了，后面你就搞不清你在说变量还是函数了。 但有时候你可能必须要用相同的名字，否则你可能会遇到一些难题。但是不管怎么说，尽量避开这种做法。

Q: 函数可以限制参数的传递个数？

> 这取决于你python的版本以及你用什么电脑，即使有这个限制的数字也是非常大的。为了使函数方便使用，实际的限制大约时5个参数，也就是说当函数参数超过5个的时候，函数就会变得不方便使用。

Q:你能在一个函数中调用另一个函数吗？

> 可以，本书后面有一个制作游戏的例子就是这么做的。

练习20.函数和文件

回忆一下函数的要点，然后一边做这节练习，一边注意一下函数和文件是如何在一起协作发挥作用的。

```
from sys import argv

script, input_file = argv

def print_all(f):
    print f.read()

def rewind(f):
    f.seek(0)

def print_a_line(line_count, f):
    print line_count, f.readline()

current_file = open(input_file)

print "First let's print the whole file:\n"

print_all(current_file)

print "Now let's rewind, kind of like a tape."

rewind(current_file)

print "Let's print three lines:"

current_line = 1
print_a_line(current_line, current_file)

current_line = current_line + 1
print_a_line(current_line, current_file)

current_line = current_line + 1
print_a_line(current_line, current_file)
```

特别注意一下，每次运行 `print_a_line` 时，我们是怎样传递当前的行号信息的。

你看到的结果

```
$ python ex20.py test.txt
First let's print the whole file:

This is line 1
This is line 2
This is line 3

Now let's rewind, kind of like a tape.
Let's print three lines:
1 This is line 1

2 This is line 2

3 This is line 3
```

附加题

> 1. 通读脚本，在每行之前加上注解，以理解脚本里发生的事情。
> 2. 每次 `print_a_line` 运行时，都传递了一个叫 `current_line` 的变量。在每次调用函数时，打印出 `current_line` 的值，跟踪一下它在 `print_a_line` 中是怎样变成 `line_count` 的。
> 3. 找出脚本中每一个用到函数的地方。检查 `def` 一行，确认参数没有用错。
> 4. 上网研究一下 `file` 中的 `seek` 函数是做什么用的。试着运行 `pydoc file` 看看能不能学到更多。
> 5. 研究一下 `+=` 这个简写操作符的作用，写一个脚本，把这个操作符用在里边试一下。

常见问题

Q:函数 `print_all` 中的 `f` 是什么?

> `f` 就是一个变量，就好像在练习18中其他的变量一样，只不过这次它代表了一个文件。Python中的文件就好像老旧的磁带驱动器，或者是像现在的DVD播放器。它有一个 "磁头"，你可以在文件中"查找"到这个磁头的位置，并且从那个位置开始运行。你每执行一次 `f.seek(0)` ，就靠近文件的开头一点。每执行一次 `f.readline()` 你就从文件中读取一行内容，并且把“磁头”移动到文件末尾，换行符 `\n` 的后面。继续学习本书，你会看到更多的解释。

Q: 文件中为什么有3个空行？

> 函数 `readline()` 返回一行以 `\n` 结尾的文件内容， 在你调用print函数的最后增加一个逗号','，用来避免为每一行添加两个换行符 `\n` 。

Q:为什么 `seek(0)` 方法没有把 `current_line` 的值修改为0？

> 首先， `seek()` 方法是按字节而不是按行为处理单元的。代码 `seek(0)` 重新定位在文件的第0位（第一个字节处）。再次， `current_line` 是一个变量，在文件中没有真正的意义可言。我们是在手动的增加它的值。

Q: `+=` 是什么?

> 你应该知道在英语里我们可以简写 "it is" 为 "it's"，简写 "you are" 为 "you're"。在英语里我们把这种写法称为缩写，同样的， `+=` 是 `=` 和 `+` 两个操作符的缩写. 比如 `x = x + y` 可以缩写为 `x += y` .

Q: `readline()` 怎么知道每一行的分界在哪里？

> `readline()`内部代码是扫描文件的每一个字节，直到找到一个`\n`字符代码，然后停止阅读，并返回到此之前获得的所有内容。代码中f的责任是在每次调用`readline()`之后，维护“磁头”在文件中的位置，以保证继续读后面的每一行。

练习21.函数的返回值

你已经学过使用 `=` 给变量命名，将变量定义为某个数字或者字符串。接下来我们将让你见证更多奇迹。我们要给你演示如何使用 `=` 以及一个新的Python关键字 `return` 来将变量设置为“一个函数的值”。有一点你需要极其注意，不过我们先来编写下面的脚本吧：

```
def add(a, b):
    print "ADDING %d + %d" % (a, b)
    return a + b

def subtract(a, b):
    print "SUBTRACTING %d - %d" % (a, b)
    return a - b

def multiply(a, b):
    print "MULTIPLYING %d * %d" % (a, b)
    return a * b

def divide(a, b):
    print "DIVIDING %d / %d" % (a, b)
    return a / b

print "Let's do some math with just functions!"

age = add(30, 5)
height = subtract(78, 4)
weight = multiply(90, 2)
iq = divide(100, 2)

print "Age: %d, Height: %d, Weight: %d, IQ: %d" % (age, height, weight, iq)

# A puzzle for the extra credit, type it in anyway.
print "Here is a puzzle."

what = add(age, subtract(height, multiply(weight, divide(iq, 2))))

print "That becomes: ", what, "Can you do it by hand?"
```

现在我们创建了自己的加减乘除数学函数： `add` , `subtract` , `multiply` , 以及 `divide` 。重要的是函数的最后一行，例如 `add` 的最后一行是 `return a + b` ，它实现的功能是这样的:

> 1. 我们调用函数时使用了两个参数： a 和 b 。
> 2. 我们打印出这个函数的功能，这里就是计算加法（adding）
> 3. 接下来我们告诉 Python 让它做某个回传的动作：我们将 `a + b` 的值返回(return)。或者你可以这么说：“我将 a 和 b 加起来，再把结果返回。”
> 4. Python 将两个数字相加，然后当函数结束的时候，它就可以将 `a + b` 的结果赋予一个变量。

和本书里的很多其他东西一样，你要慢慢消化这些内容，一步一步执行下去，追踪一下究竟发生了什么。为了帮助你理解，本节的附加题将让你解决一个迷题，并且让你学到点比较酷的东西。

你看到的结果

```
$ python ex21.py
Let's do some math with just functions!
ADDING 30 + 5
SUBTRACTING 78 - 4
MULTIPLYING 90 * 2
DIVIDING 100 / 2
Age: 35, Height: 74, Weight: 180, IQ: 50
Here is a puzzle.
DIVIDING 50 / 2
MULTIPLYING 180 * 25
SUBTRACTING 74 - 4500
ADDING 35 + -4426
That becomes:  -4391 Can you do it by hand?
```

附加题

> 1. 如果你不是很确定`return`的功能，尝试自己写几个函数出来，让它们返回一些值。你可以将任何可以放在`=`右边的东西作为一个函数的返回值。
> 2. 这个脚本的结尾是一个迷题。我将一个函数的返回值用作了另外一个函数的参数。我将它们连接一起，就像写数学等式一样。这样可能有些难懂，不过运行一下你就知道结果了。你可以试试看能不能用正常的方法实现和这个表达式一样的功能。
> 3. 一旦你解决了这个迷题，试着修改一下函数里的某些部分，然后看会有什么样的结果。你可以有目的地修改它，让它输出另外一个值。
> 4. 颠倒过来再做一次。写一个简单的等式，使用相同的函数来计算它。

这节习题可能会让你有些头大，不过慢慢来，把它当做一个小游戏，解决这样的迷题也是编程的乐趣之一。后面你还会看到类似的小谜题。

常见问题

Q:为什么Python打印公式或函数是反向的？

> 它们并不是真正的反向的, it's "inside out." When you start breaking down the function into separate formulas and function calls you'll see how it works. Try to understand what I mean by "inside out" rather than "backward."

Q: 怎样使用 `raw_input()` 输入我自己的值？

> 还记得 `int(raw_input())` 吗？这样做有一个问题就是你不能输入浮点数，不过你可以使用 `float(raw_input())` 来输入。

Q: 你说的 "写一个公式"是什么意思?

> 试试先写 `24 + 34 / 100 - 1023`，再用我们的函数转化一下。现在给你的数学公式加入变量，这样它就变成了一个公式。

练习22.到目前为止你学到了什么？

这节以及下节的练习中没有代码编写，所以也不会有习题答案以及附加题。实际上，你可以把这节练习当做一个大的附加题对待，我会带你复习一下你学到的东西。

首先，回顾你做过的每一个练习的脚本，把遇到的每一个词和符号（或者叫做字符）写下来，确保你的符号表是完整的，没有遗漏。

然后，在每个关键词或者符号后面，写下他们的名字以及作用。如果你在本书中找不到一个符号的名字，你可以上网搜索一下；如果你不知道一个关键词或者符号的作用，找到用到该字符的练习章节，通读一遍，并在脚本中测试一下它的功能。

你或许会遇到一些你怎么也找不到答案的东西，把他们记在列表中，等你下次遇到的时候，就不会轻易的跳过了。

完成你的列表之后，再花几天时间重写一遍这个列表，并确认列表的内容都是正确的。你可能会觉得这件事情很无聊，但是一定要坚持完成。

等你记住列表中的内容，尝试默写一遍，包括这些字符的名字和他们的作用，。如果发现忘记一些了某些内容，就回去再记一遍。

> **Warning:**请牢记：这节练习没有失败，只有尝试。

你学到的东西

这种记忆练习是很枯燥的，所以你要明确这种练习的重要意义：它能帮你明确目标，知道你所有努力的目的。

这节练习中，你学到的是各种符号的名称和作用，这样能帮助你更容易的阅读代码。这跟你学习英文的字母表有一样的意义，不一样的是，Python中有一些字符是你并不熟悉的。

慢慢做，别让它成为你的负担。这些符号对你来说应该比较熟悉，所以记住它们应该不是很费力的事情。你可以一次花15分钟，然后休息一下。劳逸结合可以让你学得更快，而且可以让你保持士气。

练习23.阅读代码

上一周你应该已经牢记了你的符号列表。现在你需要将这些运用起来，再花一周的时间，在网上阅读代码。这个任务初看会觉得很艰巨。我将直接把你丢到深水区呆几天，让你竭尽全力去读懂真是项目里的代码。这节练习的目的不是让你读懂所有代码，而是让你学会下面的技能：

> 1. 找到你需要的 Python 代码。
> 2. 通读代码，并找到你需要的文件。
> 3. 尝试理解你找到的代码

以你现在的水平，你还不具备完全理解你找到的代码的能力，不过通过接触这些代码，你可以熟悉真正的编程项目是什么样子。

当你做这节练习时，你可以把自己当成是一个人类学家来到了一片陌生的大陆，你只懂得一丁点本地语言，但你需要接触当地人并且生存下去。当然做练习不会碰到生存问题，这毕竟这不是荒野或者丛林。

你要做的事情如下：

> 1. 浏览器登陆 `bitbucket.org` , `github.com` , 或者 `gitorious.org` 搜索"python."
> 2. 随便找一个项目，然后点进去。
> 3. 点击 `Source` 标签，浏览目录和文件列表，直到你看到以 `.py` 结尾的文件。
> 4. 从头开始阅读你找到的代码。把它的功能用笔记记下来。
> 5. 如果你看到一些有趣的符号或者奇怪的字符，你可以把它们记下来，日后再进行研究。
>
> **Warning:** 忽略那些提到 “Python 3” 的项目，它们只会让你变迷糊。

就是这样，你的任务是使用你目前学到的东西，看自己能不能读懂一些代码，看出它们的功能来。你可以先粗略地阅读，然后再细读。也许你还可以试试将难度比较大的部分一字不漏地朗读出来。

试试其他的站点:

> - github.com
> - launchpad.net
> - gitorious.org
> - sourceforge.net

练习24.更多的练习

你离这本书第一部分的结尾已经不远了，你应该已经具备了足够的 Python 基础知识，可以继续学习一些编程的原理了，但你应该做更多的练习。这个练习的内容比较长，它的目的是锻炼你的毅力，下一个习题也差不多是这样的，好好完成它们，做到完全正确，记得仔细检查。

```
print "Let's practice everything."
print 'You\'d need to know \'bout escapes with \\ that do \n newlines and \t tabs.'

poem = """
\tThe lovely world
with logic so firmly planted
cannot discern \n the needs of love
nor comprehend passion from intuition
and requires an explanation
\n\t\twhere there is none.
"""

print "--------------"
print poem
print "--------------"

five = 10 - 2 + 3 - 6
print "This should be five: %s" % five

def secret_formula(started):
    jelly_beans = started * 500
    jars = jelly_beans / 1000
    crates = jars / 100
    return jelly_beans, jars, crates

start_point = 10000
beans, jars, crates = secret_formula(start_point)

print "With a starting point of: %d" % start_point
print "We'd have %d beans, %d jars, and %d crates." % (beans, jars, crates)

start_point = start_point / 10

print "We can also do that this way:"
print "We'd have %d beans, %d jars, and %d crates." % secret_formula(start_point)
```

你看到的结果

```
$ python ex24.py
Let's practice everything.
You'd need to know 'bout escapes with \ that do
 newlines and 	tabs.
--------------

	The lovely world
with logic so firmly planted
cannot discern
 the needs of love
nor comprehend passion from intuition
and requires an explanation

		where there is none.

--------------
This should be five: 5
With a starting point of: 10000
We'd have 5000000 beans, 5000 jars, and 50 crates.
We can also do that this way:
We'd have 500000 beans, 500 jars, and 5 crates.
```

附加题

> 1. 记得仔细检查结果，从后往前倒着检查，把代码朗读出来，在不清楚的位置加上注释。
> 2. 故意把代码改错，运行并检查会发生什么样的错误，并且确认你有能力改正这些错误。

常见问题

Q:为什么在后面你把 `jelly_beans` 这个变量叫做 `beans` ?

> 这是函数如何工作的一部分。记住，在函数中，变量都是临时的.当你返回一个变量的时候，你可以把它赋值给另一个变量。我只是定义了一个叫做‘beans’的新变量去接收返回值。

Q:你所说的反向阅读代码是什么意思?

> 从最后一行开始阅读。对比你的代码和我的是不是一样。如果确认一样，向上移动一行阅读，直到你读到脚本的第一行为止。

Q:那首诗是谁写的？

> 我写的。

练习25.更多更多的练习

我们将做一些关于函数和变量的练习，以确认你真正掌握了这些知识。这节练习对你来说可以说是：写程序，逐行研究，弄懂它。

过这节练习还是有些不同，你不需要运行它，取而代之，你需要将它导入到 python 里通过自己执行函数的方式运行。

```
def break_words(stuff):
    """This function will break up words for us."""
    words = stuff.split(' ')
    return words

def sort_words(words):
    """Sorts the words."""
    return sorted(words)

def print_first_word(words):
    """Prints the first word after popping it off."""
    word = words.pop(0)
    print word

def print_last_word(words):
    """Prints the last word after popping it off."""
    word = words.pop(-1)
    print word

def sort_sentence(sentence):
    """Takes in a full sentence and returns the sorted words."""
    words = break_words(sentence)
    return sort_words(words)

def print_first_and_last(sentence):
    """Prints the first and last words of the sentence."""
    words = break_words(sentence)
    print_first_word(words)
    print_last_word(words)

def print_first_and_last_sorted(sentence):
    """Sorts the words then prints the first and last one."""
    words = sort_sentence(sentence)
    print_first_word(words)
    print_last_word(words)
```

首先以正常的方式`python ex25.py`运行，找出里边的错误，并修正。然后你需要跟着下面的答案部分完成这节练习。

你看到的结果

在这节练习中，我们将在Python解析器中，以交互的方式和你写的`ex25.py`文件交流，你可以像下面这样在命令行中启动python解析器：

```
$ python
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

你的输出应该和我类似，在 `>` 符号之后，你可以输入并立即执行python代码。我希望你用这种方式逐行输入下方的python代码，并看看他们有什么用：

```
import ex25
sentence = "All good things come to those who wait."
words = ex25.break_words(sentence)
words
sorted_words = ex25.sort_words(words)
sorted_words
ex25.print_first_word(words)
ex25.print_last_word(words)
words
ex25.print_first_word(sorted_words)
ex25.print_last_word(sorted_words)
sorted_words
sorted_words = ex25.sort_sentence(sentence)
sorted_words
ex25.print_first_and_last(sentence)
ex25.print_first_and_last_sorted(sentence)
```

这是我做出来的样子：

```
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import ex25
>>> sentence = "All good things come to those who wait."
>>> words = ex25.break_words(sentence)
>>> words
['All', 'good', 'things', 'come', 'to', 'those', 'who', 'wait.']
>>> sorted_words = ex25.sort_words(words)
>>> sorted_words
['All', 'come', 'good', 'things', 'those', 'to', 'wait.', 'who']
>>> ex25.print_first_word(words)
All
>>> ex25.print_last_word(words)
wait.
>>> words
['good', 'things', 'come', 'to', 'those', 'who']
>>> ex25.print_first_word(sorted_words)
All
>>> ex25.print_last_word(sorted_words)
who
>>> sorted_words
['come', 'good', 'things', 'those', 'to', 'wait.']
>>> sorted_words = ex25.sort_sentence(sentence)
>>> sorted_words
['All', 'come', 'good', 'things', 'those', 'to', 'wait.', 'who']
>>> ex25.print_first_and_last(sentence)
All
wait.
>>> ex25.print_first_and_last_sorted(sentence)
All
who
```

当你写完这些代码，确保你能在`ex25.py`中找到运行的函数，并且明白它们每一个都是如何工作的。如果你的运行结果出错或者跟我的结果不一样，你就需要检查并修复你的代码，重启python解析器，再次运行程序。

附加题

> 1. 研究答案中没有分析过的行，找出它们的来龙去脉。确认自己明白了自己使用的是模块`ex25`中定义的函数。
> 2. 试着执行`help(ex25)`和`help(ex25.break_words)`。这是你得到模块帮助文档的方式。 所谓帮助文档就是你定义函数时放在`"""`之间的东西，它们也被称作`documentation comments （文档注解）`，后面你还会看到更多类似的东西。
> 3. 重复键入`ex25.`是很烦的一件事情。有一个捷径就是用`from ex25 import *`的方式导入模组。这相当于说：“我要把 ex25 中所有的东西 import 进来。”程序员喜欢说这样的倒装句，开一个新的会话，看看你所有的函数是不是已经在那里了。
> 4. 把你脚本里的内容逐行通过python编译器执行，看看会是什么样子。你可以通过输入`quit()`来退出Python。

常见问题

Q: 我的某些函数没有打印输出任何值

> 你的函数末尾可能缺少`return`语句，检查你的文件，确保每一行代码的正确性。

Q: 我输入`import ex25`的时候遇到报错`import: command not found`.

> 仔细观察“你看到的结果”部分，看我是如何运行程序的。我是在python解析器里而不是在命令行运行程序。你应该先运行python解析器。

Q: 当我输入`import ex25.py`的时候，我遇到报错`ImportError: No module named ex25.py`.

> 不要加上`.py`。Python知道文件是以`.py`结尾的，所以你只要输入`import ex25`就可以了。

Q:运行程序是，遇到报错信息`SyntaxError: invalid syntax`

> 这个信息说明在报错的这一行或之前的某一行你可能少写了一个`(` 或者 `"` 或者其它的语法错误。当你遇到这个报错的时候，从报错的行开始，向上检查是否每一行代码都是正确的。

Q:函数`words.pop`是如何改变变量`words`的值的？

> 这是一个复杂的问题,在这个实例中，`words`是一个列表，所以你可以调用它的一些命令，而它也会保留这些命令的结果。这类似于文件的工作原理。

Q: 什么情况下，我可以在函数中用`print`代替`return`？

> `return`是从函数给出的代码行调用的函数的结果。你可以把函数理解成 通过参数获取输入，并通过`return`返回输出,而`print`是与这个过程完全无关的，它只负责在终端打印输出。

练习26.恭喜你，可以进行一次考试了

你已经完成这本书的前半部分了，不过后半部分才更有趣。你将会学习逻辑，并通过条件判断实现有用的功能。

在你继续学习之前，你要完成一道试题。这道试题很难，因为它需要你修正别人写的代码。当你成为程序员以后，你将需要经常面对别的程序员的代码，也许还有他们的傲慢态度，他们会经常说自己的代码是完美的。

这样的程序员是自以为是不在乎别人的蠢货。优秀的程序员也会认为自己的代码总有出错的可能，他们会先假设是自己的代码有问题，然后用排除法清查所有可能是自己有问题的地方，最后才会得出“这是别人的错误”这样的结论。

在这节练习中，你将面对一个水平糟糕的程序员，并改好他的代码。我将习题 24 和 25 胡乱拷贝到了一个文件中，随机地删掉了一些字符，然后添加了一些错误进去。大部分的错误是Python在执行时会告诉你的，还有一些算术错误是你要自己找出来的。还有的就是格式和拼写错误了。

所有这些错误都是程序员很容易犯的，就算有经验的程序员也不例外。

这节练习中你的任务是用你所有的技能改进这个脚本。你可以先分析这个文件，或者你还可以把它像学期论文一样打印出来，修正里边的每一个缺陷，重复修正和运行的动作，直到这个脚本可以完美地运行起来。在整个过程中不要寻求帮助，如果你卡在某个地方无法进行下去，那就休息一会晚点再做。

就算你需要几天才能完成，也不要放弃，直到完全改对为止。

最后要说的是，这个练习的目的不是写程序，而是修正现有的程序，你需要访问下面的网站：

```
http://learnpythonthehardway.org/book/exercise26.txt
```

从那里把代码复制粘贴过来，命名为 `1ex26.py` ，这也是本书唯一一处允许你复制粘贴的地方。

常见问题

Q: 我必须引用 `ex25.py` 还是可以删除对它的引用？

> 两种都可以. 这个文件已经包含 `ex25.py` 的函数了，所以也可以先删除对它的引用。

Q: 当我修复这个脚本之后，我可以运行它吗？

> 当然可以。计算机就是用来帮忙的，尽可能的使用它。

练习27.记住逻辑

到此为止你已经学会了读写文件，命令行处理，以及很多 Python 数学运算功能。今天，你将要开始学习逻辑了。

你要学习的不是研究院里的高深逻辑理论，只是程序员每天都用到的让程序跑起来的基础逻辑知识。

学习逻辑之前你需要先记住一些东西。这个练习我要求你坚持一个星期，就算你烦得不得了，也要坚持下去。这个练习会让你背下来一系列的逻辑表格，这会让你更容易地完成后面的习题。

需要事先警告你的是：这件事情一开始一点乐趣都没有，一开始你会觉得它很无聊乏味，但它的目的是教会你一个程序员必备的重要技能。你必须记住一些重要的概念，一旦你明白了这些概念，你会相当有成就感，虽然一开始你会觉得它们很难掌握，就跟和乌贼摔跤一样，而等到某一天，你会刷的一下豁然开朗。你会从这些基础的记忆学习中得到丰厚的回报。

这里告诉你一个记住某样东西，而不让自己抓狂的小技巧：在一整天里，每次记忆一小部分，把你最需要加强的部分标记起来。不要想着在两小时内连续不停地背诵，这不会有什么好的效果。不管你花多长时间，你的大脑也只会留住你在前 15 或者 30 分钟内看过的东西。另外，你需要制作一些索引卡片，卡片正面写下逻辑关系，反面写下答案。你的目标是：拿出一张卡片来，看到正面的表达式，例如 “True or False”，你可以立即说出背面的结果是 “True”！坚持练习，直到你能做到这一点为止。

一旦你能做到这一点了，接下来你需要每天晚上写一份真值表出来。不要只是抄写，试着默写，如果发现哪里没记住的话，就飞快地撇一眼这里的答案。这样做可以训练你的大脑记住整个真值表。

不要在这上面花超过一周的时间，因为你在后面的应用过程中还会继续学习它们。

逻辑术语

在 python 中我们会用到下面的术语（字符或者词汇）来定义事物的真(True)或者假(False)。计算机的逻辑就是在程序的某个位置检查这些字符或者变量组合在一起表达的结果是真是假。

> - and 与
> - or 或
> - not 非
> - != (not equal) 不等于
> - == (equal) 等于
> - > = (greater-than-equal) 大于等于
>
> - <= (less-than-equal) 小于等于
> - True 真
> - False 假

其实你已经见过这些字符了，但这些词汇你可能还没见过。这些词汇(and, or, not)和你期望的效果其实是一样的，跟英语里的意思一模一样。

真值表

我们将使用下面这些字符来创建你需要记住的真值表：

NOT	TRUE
not False	True
not True	False

OR	TRUE?
True or False	True
True or True	True
False or True	True
False or False	False

AND	TRUE?
True and False	False
True and True	True
False and True	False
False and False	False

NOT OR	TRUE?
not (True or False)	False
not (True or True)	False
not (False or True)	False
not (False or False)	True

NOT AND	TRUE?
not (True and False)	True
not (True and True)	False
not (False and True)	True
not (False and False)	True

!=	TRUE?
1 != 0	True
1 != 1	False
0 != 1	True
0 != 0	False

==	TRUE?
1 == 0	False
1 == 1	True
0 == 1	False
0 == 0	True

现在使用这些表格创建你自己的卡片，再花一个星期慢慢记住它们。记住一点：这本书中没有失败，只要每天尽力去学，在尽力的基础上再多花一点功夫就可以了。

常见问题

Q:我不能只是学习布尔值的概念，而不记忆吗？

> 你当然可以这么做，但是当你编码的时候，你就需要不停的查找检索布尔值的规则。如果你先记住他们，这不仅仅是锻炼你的记忆能力，也使得这些操作更加自然。在此之后，布尔值的概念对你来说就会很简单。

练习28.布尔表达式

上一节学到的逻辑组合的正式名称是"布尔逻辑表达式(boolean logic expression)"。在编程中，布尔逻辑可以说是无处不在。它们是计算机运算的基础和重要组成部分，掌握它们就跟学音乐掌握音阶一样重要。

在这节练习中，你将在python里使用到上节学到的逻辑表达式。先为下面的每一个逻辑问题写出你认为的答案，每一题的答案要么为 `True` 要么为 `False` 。写完以后，你需要将python运行起来，把这些逻辑语句输入进去，确认你写的答案是否正确。

```
True and True
False and True
1 == 1 and 2 == 1
"test" == "test"
1 == 1 or 2 != 1
True and 1 == 1
False and 0 != 0
True or 1 == 1
"test" == "testing"
1 != 0 and 2 == 1
"test" != "testing"
"test" == 1
not (True and False)
not (1 == 1 and 0 != 1)
not (10 == 1 or 1000 == 1000)
not (1 != 10 or 3 == 4)
not ("testing" == "testing" and "Zed" == "Cool Guy")
1 == 1 and (not ("testing" == 1 or 1 == 0))
"chunky" == "bacon" and (not (3 == 4 or 3 == 3))
3 == 3 and (not ("testing" == "testing" or "Python" == "Fun"))
```

在本节结尾的地方我会给你一个理清复杂逻辑的技巧。

所有的布尔逻辑表达式都可以用下面的简单流程得到结果：

> 1. 找到相等判断的部分 (`==` 或者 `!=`)，将其改写为其最终值 (`True` 或 `False`)。
> 2. 找到括号里的 `and/or` ，先算出它们的值。
> 3. 找到每一个 `not` ，算出他们反过来的值。
> 4. 找到剩下的 `and/or` ，解出它们的值。
> 5. 等你都做完后，剩下的结果应该就是 `True` 或者 `False` 了。

下面我们以20行的逻辑表达式演示一下：

```
3 != 4 and not ("testing" != "test" or "Python" == "Python")
```

接下来你将看到这个复杂表达式是如何逐级解为一个单独结果的：

> 1. > 解出每一个等值判断:
>
> > a. `3 != 4` 为 `True`: `True and not ("testing" != "test" or "Python" == "Python")` b. `"testing" != "test"` 为 `True`: `True and not (True or "Python" == "Python")` c. `"Python" == "Python"` 为 `True`: `True and not (True or True)`
>
> 1. > 找到括号中的每一个`and/or`:
>
> > a. `(True or True)` 为 True: `True and not (True)`
>
> 1. 找到每一个 `not` 并将其逆转:> > a. `not (True)` 为 False: `True and False`
>
> 1. 找到剩下的 `and/or`,解出它们的值:> > a. `True and False` 为 `False`

这样我们就解出了它最终的值为 False.

> **Warning:**复杂的逻辑表达式一开始看上去可能会让你觉得很难。而且你也许已经碰壁过了,不过别灰心,这些“逻辑体操”式的训练只是让你逐渐习惯起来,这样后面你可以轻易应对编程里边更酷的一些东西。只要你坚持下去,不放过自己做错的地方就行了。如果你暂时不太能理解也没关系,弄懂的时候总会到来的。

你看到的结果

以下内容是在你自己猜测结果以后,通过和 python 对话得到的结果:

```
$ python
Python 2.5.1 (r251:54863, Feb  6 2009, 19:02:12)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> True and True
True
>>> 1 == 1 and 2 == 2
True
```

附加题

> 1. Python 里还有很多和`!=`、 `==` 类似的操作符. 试着尽可能多地列出 Python 中的等价运算符。例如 `<` 或者 `<=` 。
> 2. 写出每一个等价运算符的名称。例如 `!=` 叫 “not equal(不等于)”。
> 3. 在 python 中测试新的布尔操作。在敲回车前你需要喊出它的结果。不要思考,凭自己的第一感就可以了。把表达式和结果用笔写下来再敲回车,最后看自己做对多少,做错多少。
> 4. 把习题 3 那张纸丢掉,以后你不再需要查询它了。

常见问题

Q: 为什么 `"test" and "test"` 返回 `"test"` 以及 `1 and 1` 返回 `1` 而不是 `True` ?

> python和很多语言可以返回布尔表达式中的一个操作数，而不仅仅是真或假。这意味着如果你计算`False and 1` 你会得到表达式的第一个操作数 (False) ，但是如果你计算`True and 1`的时候，你得到它的第二个操作数(1)。试一试吧。

Q: `!=` 和 `<>` 有什么不同吗?

> Python已经声明赞成使用`!=`而弃用`<>`所以尽量使用`!=`吧。其他的应该没有区别了。

Q:有没有捷径去判断布尔表达式的值?

> 有的。任何的`and`表达式包含一个`False`结果就是`False`,任何`or`表达式有一个`True`结果就是`True`，你就可以在此处得到结果，但要确保你能处理整个表达式，因为后面这是一个很有用的技能。

练习29.IF 语句

下面是你要写又一个python脚本，这节练习会向你介绍“if语句”。把这段代码输入脚本，让它正常运行，看看你有没有什么收获。

```
people = 20
cats = 30
dogs = 15

if people < cats:
    print "Too many cats! The world is doomed!"

if people > cats:
    print "Not many cats! The world is saved!"

if people < dogs:
    print "The world is drooled on!"

if people > dogs:
    print "The world is dry!"

dogs += 5

if people >= dogs:
    print "People are greater than or equal to dogs."

if people <= dogs:
    print "People are less than or equal to dogs."

if people == dogs:
    print "People are dogs."
```

你看到的结果

```
$ python ex29.py
Too many cats! The world is doomed!
The world is dry!
People are greater than or equal to dogs.
People are less than or equal to dogs.
People are dogs.
```

附加题

猜猜“if语句”是什么，它有什么用处。在做下一道习题前，试着自己回答下面的问题:

> 1. 你认为 `if` 对于它下一行的代码做了什么？
> 2. 为什么 `if` 语句的下一行需要缩进？
> 3. 如果不缩进，会怎样？
> 4. 把习题 27 中的其它布尔表达式放到`if语句`中能不能运行呢？试一下。
> 5. 如果把变量 `people` , `cats` , 和 `dogs` 的初始值改掉，会怎样？

常见问题

Q: `+=` 表示什么意思？

> 代码`x += 1`和`x = x + 1` 实现的是一样的功能，但是可以少输入一些字符。你可以称之为“增量”操作符。`-=` 也是相同的，后面你会看到更多的相关解释。

练习30.Else 和 If

上一节习题中你写了一些“if语句(if-statements)”，并且试图猜出它们实现的是什么功能。在你继续学习之前，我给你解释一下上一节的附加题的答案。上一节的附加习题你做过了吧

> 1. 认为 `if` 对于它下一行的代码做了什么?`If语句`为代码创建了一个所谓的“分支”，这有点像选择自己毛线的书籍，你做了选择会打开一个页面，如果做了另一个选择，会到一个不同的方向。`if`语句告诉你的脚本：“如果这个布尔表达式是真的，就执行它下面的语句，否则就跳过这段代码”。
>
> 1. 为什么 if 语句的下一行需要缩进？代码的最后又一个冒号“：”，是告诉python要创建一个新代码块的方式，缩进4个空格，是标志那些代码属于这个代码块。这和你在本书的上半部分中定义函数的做法是一样的。
>
> 1. 如果不缩进，会怎样？如果没有缩进，你的代码将会报错，Python需要你在输入一行以冒号结尾的代码后有缩进。
>
> 1. 把习题 27 中的其它布尔表达式放到if语句中能不能运行呢？试一下。 可以。而且不管多复杂都可以，虽然写复杂的东西并不是一种好的编程风格。
>
> 1. 如果把变量`people`,`cats`,和`dogs`的初始值改掉，会怎样？因为你比较的对象是数字，如果你把这些数字改掉的话，某些位置的`if`语句会被演绎为 True，而它下面的代码区段将被运行。你可以试着修改这些数字，然后在头脑里假想一下那一段代码会被运行。

对比咱们的答案，确认自己真正懂得“代码块”的含义。这点对于你下一节的练习很重要，因为你将会写很多的 if 语句。

把下面这段写下来，并让它运行起来：

```
people = 30
cars = 40
trucks = 15

if cars > people:
    print "We should take the cars."
elif cars < people:
    print "We should not take the cars."
else:
    print "We can't decide."

if trucks > cars:
    print "That's too many trucks."
elif trucks < cars:
    print "Maybe we could take the trucks."
else:
    print "We still can't decide."

if people > trucks:
    print "Alright, let's just take the trucks."
else:
    print "Fine, let's stay home then."
```

你看到的结果

```
$ python ex30.py
We should take the cars.
Maybe we could take the trucks.
Alright, let's just take the trucks.
```

附加题

> 1. 想一下 `elif` 和 `else` 的功能。
> 2. 将 `cars` , `people` , 和 `buses` 的数量改掉，然后追溯每一个 `if` 语句。看看最后会打印出什么来。
> 3. 试着写一些复杂的布尔表达式，例如 `cars > people` 和 `buses < cars` 等。
> 4. 给每一行加上注释，解释每一句代码是什么功能。[1]

常见问题

Q: 如果多个 `elif` 块为真，会怎样？

> Python的启动和运行只会针对第一个为真的代码块，所以你说的那种情况，只会执行第一块。

练习31.做出决定

这本书的上半部分你打印了一些东西，而且调用了函数，不过一切都是直线式进行的。你的脚本从最上面一行开始，一路运行到结束，但其中并没有决定程序流向的分支点。现在你已经学了 `if` , `else` ,和 `elif` ，你就可以开始创建包含条件判断的脚本了。

上一个脚本中你写了一系列的简单提问测试。这节的脚本中，你将需要向用户提问，依据用户的答案来做出决定。把脚本写下来，多多鼓捣一阵子，看看它的工作原理是什么。

```
print "You enter a dark room with two doors.  Do you go through door #1 or door #2?"

door = raw_input("> ")

if door == "1":
    print "There's a giant bear here eating a cheese cake.  What do you do?"
    print "1\. Take the cake."
    print "2\. Scream at the bear."

    bear = raw_input("> ")

    if bear == "1":
        print "The bear eats your face off.  Good job!"
    elif bear == "2":
        print "The bear eats your legs off.  Good job!"
    else:
        print "Well, doing %s is probably better.  Bear runs away." % bear

elif door == "2":
    print "You stare into the endless abyss at Cthulhu's retina."
    print "1\. Blueberries."
    print "2\. Yellow jacket clothespins."
    print "3\. Understanding revolvers yelling melodies."

    insanity = raw_input("> ")

    if insanity == "1" or insanity == "2":
        print "Your body survives powered by a mind of jello.  Good job!"
    else:
        print "The insanity rots your eyes into a pool of muck.  Good job!"

else:
    print "You stumble around and fall on a knife and die.  Good job!"
```

这里的重点是你可以在“if语句”内部再放一个“if语句”。这是一个很强大的功能，可以用来创建嵌套(nested)，其中的一个分支将引向另一个分支的子分支。

你需要理解 `if语句` 包含 `if语句` 的概念。做一下附加题，确保自己真正理解了它们。

你看到的结果

下面是我玩这个小游戏的结果，我玩的不怎么样：

```
$ python ex31.py
You enter a dark room with two doors.  Do you go through door #1 or door #2?
>  1
There's a giant bear here eating a cheese cake.  What do you do?
1\. Take the cake.
2\. Scream at the bear.
>  2
The bear eats your legs off.  Good job!
```

附加题

> 1. 为游戏添加新的部分，改变玩家做决定的位置。尽自己的能力扩展这个游戏，不过别把游戏弄得太怪异了。
> 2. 写一个全新的游戏，你可能不喜欢我提供的这个，那么自己写一个玩玩。这是你的电脑，你可以用它做任何自己想做的事情。

常见问题

Q: 可以用 `if-else` 替换 `elif` 吗？

> 某些情况下可以，但是这个也依赖于每一个`if/else`是怎么写的 。这也意味着， Python 会检查每个if-else的组合，而不是只检查`if-elif-else`组合中的第一个为假的分支，尝试用两种方式多编写一些代码，以找出他们的不同点。

Q:我怎么知道一个数字是在一个数字范围之间？

> 有两种方法：一种经典的方式是使用`0 < x < 10` 或者 `1 <= x < 10`,另一中方式是使用`x in range(1, 10)`。

Q: 怎样才能在 `if-elif-else` 代码块中增加更多的选择？

> 为每一个可能的选择增加一个`elif` 代码块。

练习32.循环和列表

现在你应该有能力写更有趣的程序出来了。如果你能一直跟得上，你应该已经看出将“if 语句”和“布尔表达式”结合起来可以让程序作出一些智能化的事情。

out.我们的程序还需要能快速的完成很多重复的工作。这节习题，我们将使用 `for循环` 来创建并打印一些列表。在练习的过程中，你会逐渐明白它们是怎么回事，我不会告诉你答案的，你要自己去找出来。

在你开始使用 for 循环之前，你需要在某个位置存放循环的结果。最好的方法是使用列表（list），顾名思义，列表就是一个按顺序存放东西的容器。它并不复杂，你只是要学习一点新的语法。首先我们看看如何创建列表：

```
hairs = ['brown', 'blond', 'red']
eyes = ['brown', 'blue', 'green']
weights = [1, 2, 3, 4]
```

你要做的是以 `[` （左方括号）开头“打开”列表，然后写下你要放入列表的东西，用逗号隔开，就跟函数的参数一样，最后用 `]` （右方括号）结束列表的定义。然后Python接收这个列表以及里边所有的内容，将其赋给一个变量。

> **Warning:**对于不会编程的人来说这是一个难点。习惯性思维告诉你的大脑大地是平的。记得上一个练习中的if语句嵌套吧？你可能觉得要理解它有些难度，因为生活中一般人不会去像这样的问题，但这样的问题在编程中几乎到处都是。你会看到一个函数调用另外一个包含if语句的函数，其中又有嵌套列表的列表。如果你看到这样的东西一时无法弄懂，就用纸笔记下来，手动分割代码，直到弄懂为止。

现在我们将使用循环创建一些列表，然后将它们打印出来:

```
the_count = [1, 2, 3, 4, 5]
fruits = ['apples', 'oranges', 'pears', 'apricots']
change = [1, 'pennies', 2, 'dimes', 3, 'quarters']

# this first kind of for-loop goes through a list
for number in the_count:
    print "This is count %d" % number

# same as above
for fruit in fruits:
    print "A fruit of type: %s" % fruit

# also we can go through mixed lists too
# notice we have to use %r since we don't know what's in it
for i in change:
    print "I got %r" % i

# we can also build lists, first start with an empty one
elements = []

# then use the range function to do 0 to 5 counts
for i in range(0, 6):
    print "Adding %d to the list." % i
    # append is a function that lists understand
    elements.append(i)

# now we can print them out too
for i in elements:
    print "Element was: %d" % i
```

你看到的结果

```
$ python ex32.py
This is count 1
This is count 2
This is count 3
This is count 4
This is count 5
A fruit of type: apples
A fruit of type: oranges
A fruit of type: pears
A fruit of type: apricots
I got 1
I got 'pennies'
I got 2
I got 'dimes'
I got 3
I got 'quarters'
Adding 0 to the list.
Adding 1 to the list.
Adding 2 to the list.
Adding 3 to the list.
Adding 4 to the list.
Adding 5 to the list.
Element was: 0
Element was: 1
Element was: 2
Element was: 3
Element was: 4
Element was: 5
```

附加题

> 1. 注意一下`range`的用法。查一下 `range` 函数并理解它。
> 2. 在第 22 行，你是否可以直接将`elements`赋值为`range(0,6)`，而无需使用 for 循环？
> 3. 在 Python 文档中找到关于列表的内容，仔细阅读以下，除了 append 以外列表还支持哪些操作？

常见问题

Q: 如何定义一个两层（2D）的列表？

> 就是一个列表在另一个列表里面，比如`[[1,2,3],[4,5,6]]`

Q: 列表和数组不是同一种东西吗？

> 依赖于语言和实现方式。在经典设计角度，由于数组列表的实现方式不同，数组列表是非常不同的。在Ruby中程序员称之为数组。在Python中,他们称之为列表。因为现在是Python调用它们，所以我们就称呼它为列表。

Q: 为什么for 循环可以使用一个没有定义过的变量？

> 在for循环开始的时候，就会定义这个变量， 并初始化。

Q: 为什么 `for i in range(1, 3):` 只循环了两次？

> `range()`函数循环的次数不包括最后一个。所以`range(1,3)`只循环到2,这是这种循环最常用的方法。

Q: `elements.append()` 实现什么功能？

> 它能实现在列表的末尾追加一个元素。打开Python解析器，自己写一个列表做些实验。当你遇到这类问题的时候，都可以在Python的解析器中做些实验，自己找到问题的答案。

练习33.while循环

接下来是一个更在你意料之外的概念：`while循环（while-loop）`。while循环会一直执行它下面的代码片段，直到它对应的布尔表达式为`False`时才会停下来。

你还能跟得上这些术语吧？如果你的某一行是以`:`（冒号）结尾，那就意味着接下来的内容是一个新的代码片段，新的代码片段是需要缩进的。只有将代码用这样的方式格式化，Python才能知道你的目的。如果你不太明白这一点，就回去看看"if 语句"和"函数"的章节，直到你明白为止。

接下来的练习将训练你的大脑去阅读这些结构化的代码。这和我们将布尔表达式烧录到你的大脑中的过程有点类似。

回到`while`循环，它所作的和`if`语句类似，也是去检查一个布尔表达式的真假，不一样的是它下面的代码片段不是只被执行一次，而是执行完后再调回到`while`所在的位置，如此重复进行，直到`while`表达式为`False`为止。

`While`循环有一个问题，那就是有时它永远不会结束。如果你的目的是循环到宇宙毁灭为止，那这样也挺好的，不过其他的情况下你的循环总需要有一个结束。

为了避免这样的问题，你需要遵循下面的规定：

> 1. 尽量少用`while-loop`，大部分时候`for-loop`是更好的选择。
> 2. 重复检查你的`while`语句，确定你的布尔表达式最终会变成`False`。
> 3. 如果不确定，就在while循环的结尾打印出你测试的值。看看它的变化。

在这节练习中，你将通过上面的三个检查学会`while-loop`：

```
i = 0
numbers = []

while i < 6:
    print "At the top i is %d" % i
    numbers.append(i)

    i = i + 1
    print "Numbers now: ", numbers
    print "At the bottom i is %d" % i

print "The numbers: "

for num in numbers:
    print num
```

你看到的结果

```
$ python ex33.py
At the top i is 0
Numbers now:  [0]
At the bottom i is 1
At the top i is 1
Numbers now:  [0, 1]
At the bottom i is 2
At the top i is 2
Numbers now:  [0, 1, 2]
At the bottom i is 3
At the top i is 3
Numbers now:  [0, 1, 2, 3]
At the bottom i is 4
At the top i is 4
Numbers now:  [0, 1, 2, 3, 4]
At the bottom i is 5
At the top i is 5
Numbers now:  [0, 1, 2, 3, 4, 5]
At the bottom i is 6
The numbers:
0
1
2
3
4
5
```

附加题

> 1. 将这个`while`循环改成一个函数，将测试条件`(i < 6)`中的6换成一个变量。
> 2. 使用这个函数重写你的脚本，并用不同的数字进行测试。
> 3. 为函数添加另外一个参数，这个参数用来定义第8行的加值`+1`，这样你就可以让它加任意值了。
> 4. 再使用该函数重写一遍这个脚本。看看效果如何。
> 5. 使用`for-loop`和`range`把这个脚本再写一遍。你还需要中间的加值操作吗？如果不去掉它，会有什么样的结果？

很有可能你会碰到程序跑着停不下来了，这时你只要按着 CTRL 再敲 c (CTRL-c)，这样程序就会中断下来了。

常见问题

Q: for 循环和while循环有什么区别？

> for 循环只能对某种事物的集合做循环，而while可以进行任何种类的循环。但是，while循环很容易出错，大部分情况for循环也是一个很好的选择。

Q: 循环好难啊，我怎么才能掌握它？

> 人们不理解循环的主要原因是因为他们不理解代码的“跳跃性”。当一个循环运行的时候，它会执行完循环的代码块，然后从代码块的末尾跳到开头。想象一下，在循环中放一些打印语句，当Python运行的时候，看一下变量在这些位置是如何变化的。把打印语句写在循环之前，循环的开头，循环的中间，以及循环结束的位置，研究一下这些输出，再试着理解一下代码是如何跳跃的。

练习34.访问列表元素

列表的用处很大，但只有你能访问里边的内容时它才能发挥出作用来。你已经学会了按顺序读出列表的内容，但如果你要得到第5个元素该怎么办呢？你需要知道如何访问列表中的元素。访问第一个元素的方法是这样的：

```
animals = ['bear', 'tiger', 'penguin', 'zebra']
bear = animals[0]
```

你定义了一个 animals 的列表，然后你用 0 来获取第一个元素?! 这是怎么回事？因为数学里边就是这样，所以 Python 的列表也是从 0 开始的。虽然看上去很奇怪，这样定义其实有它的好处，实际上设计成 0 或者 1 开头其实都可以。

最好的解释方式是将你平时使用数字的方式和程序员使用数字的方式做对比。

假设你在观看上面列表中的四种动物（ `['bear', 'tiger', 'penguin', 'zebra']` ）赛跑。 而它们比赛的名词正好跟列表里的次序一样。这是一场很激动人心的比赛，因为这些动物没打算吃掉对方，而且比赛还真的举办起来了。结果你的朋友来晚了，他想知道谁赢了比赛，他会问你"嘿，谁是第 0 名"吗？不会的，他会问"嘿，谁是第 1 名？"

这是因为动物的次序是很重要的。没有第一个就没有第二个，没有第二个也没有第三个。第零个是不存在的，因为零的意思是什么都没有。"什么都没有"怎么赢比赛嘛，完全不合逻辑。这样的数字我们称之为"序数(ordinal number)"，因为它们表示的是事物的顺序。

而程序员不能用这种方式思考问题，因为他们可以从列表的任何一个位置取出一个元素来。对程序员来说，上述的列表更像是一叠卡片如果他们想要 tiger，就抓它出来，如果想要 zebra，也一样抓取出来。要随机地抓取列表里的内容，列表的每一个元素都应该有一个地址，或者一个"index（索引）"，而最好的方式是使用以0开头的索引。相信我说的这一点吧，这种方式获取元素会更容易。这类的数字被称为"基数(cardinal number)"，它意味着你可以任意抓取元素，所以我们需要一个 0 号元素。

那么，这些知识对于你的列表操作有什么帮助呢？很简单，每次你对自己说"我要第 3 只动物"时，你需要将"序数"转换成"基数"，只要将前者减 1 就可以了。第 3 只动物的索引是2，也就是penguin。由于你一辈子都在跟序数打交道，所以你需要用这种方式来获得基数，只要减 1 就都搞定了。记住: ordinal == 有序，以 1 开始；cardinal == 随机选取, 以 0 开始。

让我们练习一下。定义一个动物列表，然后跟着做后面的练习，你需要写出所指位置的动物名称。如果我用的是"1st,2nd"等说法，那说明我用的是序数，所以你需要减去 1。如果我给你的是基数（0, 1, 2），你只要直接使用即可。

```
animals = ['bear', 'python', 'peacock', 'kangaroo', 'whale', 'platypus']
```

```
The animal at 1.
The third (3rd) animal.
The first (1st) animal.
The animal at 3.
The fifth (5th) animal.
The animal at 2.
The sixth (6th) animal.
The animal at 4.
```

对于上述每一条，以这样的格式写出一个完整的句子：“The 1st animal is at 0 and is a bear.” 然后倒过来念：“The animal at 0 is the 1st animal and is a bear.”

使用 python 检查你的答案。

附加题

> 1. 以你对于这些不同的数字类型的了解，解释一下为什么 “January 1, 2010” 里是 2010 而不是 2009？（提示：你不能随机挑选年份。）
> 2. 再写一些列表，用一样的方式作出索引，确认自己可以在两种数字之间互相翻译。
> 3. 使用 python 检查自己的答案。
>
> **Warning:** 会有程序员告诉你让你去阅读一个叫“Dijkstra”的人写的关于数字的话题。我建议你还是不读为妙。除非你喜欢听一个在编程这一行刚兴起时就停止从事编程的人对你大喊大叫。

练习35.分支和函数

你已经学会了 if 语句、函数、还有列表。现在你要练习扭转一下思维了。把下面的代码写下来，看你是否能弄懂它实现的是什么功能。

```
from sys import exit

def gold_room():
    print "This room is full of gold.  How much do you take?"

    choice = raw_input("> ")
    if "0" in choice or "1" in choice:
        how_much = int(choice)
    else:
        dead("Man, learn to type a number.")

    if how_much < 50:
        print "Nice, you're not greedy, you win!"
        exit(0)
    else:
        dead("You greedy bastard!")

def bear_room():
    print "There is a bear here."
    print "The bear has a bunch of honey."
    print "The fat bear is in front of another door."
    print "How are you going to move the bear?"
    bear_moved = False

    while True:
        choice = raw_input("> ")

        if choice == "take honey":
            dead("The bear looks at you then slaps your face off.")
        elif choice == "taunt bear" and not bear_moved:
            print "The bear has moved from the door. You can go through it now."
            bear_moved = True
        elif choice == "taunt bear" and bear_moved:
            dead("The bear gets pissed off and chews your leg off.")
        elif choice == "open door" and bear_moved:
            gold_room()
        else:
            print "I got no idea what that means."

def cthulhu_room():
    print "Here you see the great evil Cthulhu."
    print "He, it, whatever stares at you and you go insane."
    print "Do you flee for your life or eat your head?"

    choice = raw_input("> ")

    if "flee" in choice:
        start()
    elif "head" in choice:
        dead("Well that was tasty!")
    else:
        cthulhu_room()

def dead(why):
    print why, "Good job!"
    exit(0)

def start():
    print "You are in a dark room."
    print "There is a door to your right and left."
```

```
    print "Which one do you take?"

    choice = raw_input("> ")

    if choice == "left":
        bear_room()
    elif choice == "right":
        cthulhu_room()
    else:
        dead("You stumble around the room until you starve.")

start()
```

你看到的结果

这是我玩这个游戏的过程：

```
$ python ex35.py
You are in a dark room.
There is a door to your right and left.
Which one do you take?
>  left
There is a bear here.
The bear has a bunch of honey.
The fat bear is in front of another door.
How are you going to move the bear?
>  taunt bear
The bear has moved from the door. You can go through it now.
>  open door
This room is full of gold.  How much do you take?
>  1000
You greedy bastard! Good job!
```

附加题

> 1. 把这个游戏的地图画出来，把自己的路线也画出来。
> 2. 改正你所有的错误，包括拼写错误。
> 3. 为你不懂的函数写注释。
> 4. 为游戏添加更多元素。通过怎样的方式可以简化并且扩展游戏的功能呢？
> 5. 这个`gold_room`游戏使用了奇怪的方式让你键入数字。这种方式会导致什么样的bug？你可以用比检查0、1更好的方式判断输入是否是数字吗？`int()`这个函数可以给你一些头绪。

常见问题

Q: 求助！这个程序是怎样运行的？

> 当你理解一段代码遇到困难的时候，可以给每一行代码加上一段简单的注释用来解释每一句实现什么功能。尽量使你的注释短小且与代码近似.然后用图解或者写一段描述来弄懂代码是如何工作的。如果你这么做了，你就能弄明白这段代码是如何工作的。

Q: 你为什么用了 `while True` ？

> 这么写可以创建一个无限循环

Q: `exit()` 是干什么的？

> 在许多操作系统中可以使用 `exit(0)` 来中止程序，传递的数字参数表示是否遇到异常。如果使用 `exit(1)` 退出将会出现一个错误，但是用 `exit(0)` 就是正常的退出。参数部分和正常的布尔逻辑正好是相反的 (正常的布尔逻辑中 0==False) 您可以使用不同的数字来表示不同的错误结果。你也可以用 `exit(100)` 来表示一个不同于 `exit(2)` 和 `exit(1)` 的错误信息.

Q: 为什么有时候把 `raw_input` 写成 `raw_input('>')`

> `raw_input` 的参数只是一个字符串，会打印显示在要求用户输入之前。

练习36.设计和调试

现在你已经学会了`if 语句`，我将给你一些使用`for 循环`和`while 循环`的规则，以免你日后碰到麻烦。我还会教你一些调试的小技巧，以便你能发现自己程序的问题。最后，你将需要设计一个和上节类似的小游戏，不过内容略有更改。

IF 语句的规则：

> 1. 每一个“if 语句”必须包含一个 else.
> 2. 如果这个`else`永远都不应该被执行到，因为它本身没有任何意义，那你必须在else语句后面使用一个叫做`die`的函数，让它打印出错误信息,这和上一节的习题类似，这样你可以找到很多的错误。
> 3. “if 语句”的嵌套不要超过 2 层，最好尽量保持只有 1 层。
> 4. 将“if 语句”当做段落来对待，其中的每一个`if-elif-else`组合就跟一个段落的句子一样。在这种组合的最前面和最后面留一个空行以作区分。
> 5. 你的布尔测试应该很简单，如果它们很复杂的话，你需要将它们的运算事先放到一个 变量里，并且为变量取一个好名字。

如果你遵循以上规则，你就会写出比大部分程序员都好的代码来。回到上一节练习，看看我有没有遵循这些规则，如果没有的话，就将其改正过来。

> **Warning:**在日常编程中不要死板的遵守规则。在训练中，你需要通过这些规则的应用来巩固你学到的知识，而在实际编程中这些规则有时其实很蠢。如果你觉得哪个规则很蠢，就别使用它。

循环的规则

> 1. 只有在循环永不停止时使用“while循环”，这意味着你可能永远都用不到。这条只有 Python 中成立，其他的语言另当别论。
> 2. 其他类型的循环都使用“for循环”，尤其是在循环的对象数量固定或者有限的情况下。

调试的小技巧

> 1. 不要使用 “debugger”。`Debugger`所作的相当于对病人的全身扫描。你不会得到某方面的有用信息，而且你会发现它输出的信息大部分没有用，或者只会让你更困惑。
> 2. 最好的调试程序的方法是使用`print`,在各个你想要检查的关键环节将关键变量打印出来，从而检查哪里是否有错。
> 3. 让程序一部分一部分地运行起来。不要等一个很长的脚本写完后才去运行它。写一点，运行一点，修改一点。

家庭作业

写一个和上节类似的小游戏。任何题材的游戏都可以。尽量花一周的时间让这个游戏有趣一些，作为附加题，你可以尽量多的使用列表，函数和模块（还记得练习13吗？），而且，尽量弄一些新的Python代码让你的游戏运行起来。

在你开始编码之前，你应该先画一张地图出来，提前设计出玩家可能遇到的房间、怪物以及陷阱等。

当你画好了梯度，你就可以开始编码了。如果你发现地图有问题的话，修改一下，让代码和地图相匹配。

完成一个软件的最好方式是把它们拆解为像下面这样的小块：

> 1. 在纸上写下你完成这个软件所需要做的所有任务。这就是你的待办事项列表。
> 2. 先找到你列表中最容易的事情。
> 3. 在你的源代码中增加注释，作为你完成这项任务的指南。
> 4. 在这些注释下面，开始编码。
> 5. 然后立即运行你的代码，看它是否正常工作。
> 6. 循环的进行代码编写，测试运行，以及代码修正，直到代码正常运行。
> 7. 在你的列表中划掉刚完成的任务，然后再挑选下一个最容易完成的任务，重复进行以上步骤。

这套程序会帮助你在写代码的时候保持系统的、一致的风格。当你开始工作的时候，更新你的任务清单，增加你要做的，并删除已完成的。

练习37.复习符号

现在该复习你学过的符号和python关键字了，而且你在本节还会学到一些新的东西。我在这里所作的是将所有的Python符号和关键字列出来，这些都是值得掌握的重点。

在这节课中，你需要复习每一个关键字，回想它的作用并且写下来，接着上网搜索它真正的功能。有些内容可能是难以搜索的，所以这对你可能有些难度，不过无论如何，你都要尝试一下。

如果你发现记忆中的内容有误，就在索引卡片上写下正确的定义，试着将自己的记忆纠正过来。

最后，将每一种符号和关键字用在程序里，你可以用一个小程序来做，也可以尽量多写一些程序来巩固记忆。这里的关键点是明白各个符号的作用，确认自己没搞错，如果搞错了就纠正过来，然后将其用在程序里，并且通过这样的方式加深自己的记忆。

关键字

KEYWORD	DESCRIPTION	EXAMPLE
`and`	逻辑与	`True and False == False`
as	with-as 语句的一部分	with X as Y: pass
`assert`	声明	`assert False, "Error!"`
break	停止整个循环	while True: break
`class`	定义一个类	`class Person(object)`
continue	停止这一次循环，但继续下一次循环	while True: continuev
`def`	定义一个函数	`def X(): pass`
del	从字典中删除	del X[Y]
`elif`	Else if 条件	`if: X; elif: Y; else: J`
else	Else 条件	if: X; elif: Y; else: J
`except`	如果捕获异常，执行该代码块	`except ValueError, e: print e`
exec	将字符串作为Python代码执行	exec 'print "hello"'
`finally`	不管是否有异常，finally代码块都执行	`finally: pass`
for	for循环	for X in Y: pass
`from`	从某一模块中引入特定部分	`import X from Y`

`global`	定义一个全局变量	`global X`
`if`	If 条件	`if: X; elif: Y; else: J`
`import`	引入一个模块到当前模块	`import os`
`in`	for循环的一部分/ 测试 `X in Y` .	`for X in Y: pass` / `1 in [1] == True`
`is`	类似 `==` ，判断相等	`1 is 1 == True`
`lambda`	创建一个无名函数	`s = lambda y: y ** y; s(3)`
`not`	逻辑非	`not True == False`
`or`	逻辑或	`True or False == True`
`pass`	该代码块为空	`def empty(): pass`
`print`	打印一个字符串	`print 'this string'`
`raise`	代码出错时，抛出一个异常	`raise ValueError("No")`
`return`	退出函数并返回一个返回值	`def X(): return Y`
`try`	尝试代签代码块，有异常则进入except代码块	`try: pass`
`while`	While循环	`while X: pass`
`with`	一个变量的别名	`with X as Y: pass`
`yield`	暂停， 返回给调用者	`def X(): yield Y; X().next()`

数据类型

针对每一种数据类型，都举出一些例子来，例如针对string，你可以举出一些字符串，针对number，你可以举出一些数字。

TYPE	DESCRIPTION	EXAMPLE
`True`	True 布尔值.	`True or False == True`
`False`	False 布尔值.	`False and True == False`
`None`	表示 "nothing" 或者"no value".	`x = None`
`strings`	字符串，储存文本信息	`x = "hello"`
`numbers`	储存整数	`i = 100`
`floats`	储存小数	`i = 10.389`
`lists`	储存某种东西的列表	`j = [1,2,3,4]`
`dicts`	储存某些东西的键值对	`e = {'x': 1, 'y': 2}`

字符串转义序列

对于字符串转义序列，你需要在字符串中应用它们，确认自己清楚地知道它们的功能。

ESCAPE	DESCRIPTION
\	斜线
\'	单引号
\"	双引号
\a	Bell
\b	退格
\f	Formfeed
\n	换行
\r	Carriage
\t	Tab键
\v	垂直的tab

字符串格式化

ESCAPE	DESCRIPTION	EXAMPLE
`%d`	格式化整数 (不包含浮点数).	`"%d" % 45 == '45'`
`%i`	与%d相同	`"%i" % 45 == '45'`
`%o`	8进制数字	`"%o" % 1000 == '1750'`
`%u`	负数	`"%u" % -1000 == '-1000'`
`%x`	小写的十六进制数字	`"%x" % 1000 == '3e8'`
`%X`	大写的十六进制数字	`"%X" % 1000 == '3E8'`
`%e`	小写 'e'的指数标记	`"%e" % 1000 == '1.000000e+03'`
`%E`	大写 'e'的指数标记	`"%E" % 1000 == '1.000000E+03'`
`%f`	浮点数	`"%f" % 10.34 == '10.340000'`
`%F`	与%f相同	`"%F" % 10.34 == '10.340000'`
`%g`	%f 或者 %e中较短的一个	`"%g" % 10.34 == '10.34'`
`%G`	%F 或者 %E中较短的一个	`"%G" % 10.34 == '10.34'`
`%c`	字符格式化	`"%c" % 34 == '"'`
`%r`	类型格式化	`"%r" % int == "<type 'int'>"`
`%s`	字符串格式	`"%s there" % 'hi' == 'hi there'`
`%%`	表示百分号%	`"%g%%" % 10.34 == '10.34%'`

操作符

有些操作符号你可能还不熟悉，不过还是一一看过去，研究一下它们的功能，如果你研究不出来也没关系，记录下来日后解决。

OPERATOR	DESCRIPTION	EXAMPLE
`+`	加	`2 + 4 == 6`
`-`	减	`2 - 4 == -2`
`*`	乘	`2 * 4 == 8`
`**`	幂乘	`2 ** 4 == 16`
`/`	除	`2 / 4.0 == 0.5`
`//`	整除，得到除法的商。	`2 // 4.0 == 0.0`
`%`	模除，返回除法的余数。	`2 % 4 == 2`
`<`	小于	`4 < 4 == False`
`>`	大于	`4 > 4 == False`
`<=`	小于等于	`4 <= 4 == True`
`>=`	大于等于	`4 >= 4 == True`
`==`	等于，比较操作对象是否相等。	`4 == 5 == False`
`!=`	不等于	`4 != 5 == True`
`<>`	不等于	`4 <> 5 == True`
`( )`	括号	`len('hi') == 2`
`[ ]`	列表括号	`[1,3,4]`
`{ }`	字典括号	`{'x': 5, 'y': 10}`
`@`	装饰符	`@classmethod`
`,`	逗号	`range(0, 10)`
`:`	冒号	`def X():`
`.`	Dot	`self.x = 10`
`=`	赋值等于	`x = 10`
`;`	分号	`print "hi"; print "there"`
`+=`	加等于	`x = 1; x += 2`
`-=`	减等于	`x = 1; x -= 2`
`*=`	乘等于	`x = 1; x *= 2`
`/=`	除等于	`x = 1; x /= 2`
`//=`	整除等于	`x = 1; x //= 2`
`%=`	模除等于	`x = 1; x %= 2`
`**=`	幂乘等于	`x = 1; x **= 2`

花一个星期学习这些东西，如果你能提前完成就更好了。我们的目的是覆盖到所有的符号 类型，确认你已经牢牢记住它们。另外很重要的一点是这样你可以找出自己还不知道哪些 东西，为自己日后学习找到一些方向。

读代码

找一些python的代码读读试试。你可以读任何的python代码，并且可以借鉴其中的一些思想。你已经具备足够的知识去阅读代码，但是你可能还不能完全明白代码实现了什么功能。这节练习就是教给你如何用你学过的知识弄明白别人的代码。

首先，把你找到的代码打印出来，是的，你需要把它们打印出来，因为相比电脑屏幕，你的大脑和眼睛更容易看清楚纸上的内容。

接下来，通读你打印的代码，按照下面说的做一些笔记：

> 1. 找出所有的函数，以及它们的功能。
> 2. 每一个变量在哪里被赋予初始值。
> 3. 代码的不同地方有没有相同名字的变量，这可能会带来隐患。
> 4. 有没有if语句没有else代码块的，这么写对吗？
> 5. 有没有无终止的while循环
> 6. 标记出不管任何原因，你看不懂的代码部分。

第三步，当你做完上面内容之后，尝试给自己解释一下自己写的注释。说明这些函数是如何应用的，包含哪些变量，以及你想弄明白的其他事情。

最后，在所有难以理解的部分，逐行、逐个函数的跟踪每个变量的值。你也可以在准备一份打印的代码，在空白处写下你要跟踪的每个变量的值。

当你弄明白这段代码是做什么的之后，回到电脑上再读一遍代码，看看能不能找到一些新的东西。多找一些代码练习，直到你能不需要打印代码就能弄懂它们的功能为止。

附加题

> 1. 弄明白“流程图”是什么，试着画几个出来
> 2. 读代码的过程，如果发现了什么错误，尝试着改正它，并将你修改后的结果发给代码的作者。
> 3. 另一个技巧是用 `#` 给你正在读的代码加注释，有时候，你的这些注释会帮到后面来读代码的人哦。

常见问题

Q: `%d` 和 `%i` 有什么区别？

> 没有区别，只不过由于历史原因，人们更喜欢用 `%d` 。

Q: 我们怎么在网上搜索这些符号和关键字？

> 只要把“python”放在你要搜索的内容之前就可以了，比如，你想搜索 `yield` ，那么就输入 `python yield` 。

练习38.列表操作

你已经学过了列表。在你学习“while循环”的时候，你对列表进行过“追加(append)”操作，而且将列表的内容打印了出来。另外你应该还在附加题里研究过 Python 文档，看了列表支持的其他操作。这已经是一段时间以前了，所以如果你不记得了的话，就回到本书的前面再复习一遍把。

找到了吗？还记得吗？很好。那时候你对一个列表执行了`append`函数。不过，你也许还没有真正明白发生的事情，所以我们再来看看我们可以对列表进行什么样的操作。

当你看到像 mystuff.append('hello') 这样的代码时，你事实上已经在 Python 内部激发了一个连锁反应。以下是它的工作原理：

> 1. 1.Python 看到你用到了`mystuff`，于是就去找到这个变量。也许它需要倒着检查看你有没有在哪里用`=`创建过这个变量，或者检查它是不是一个函数参数，或者看它是不是一个全局变量。不管哪种方式，它得先找到 mystuff 这个变量才行。
> 2. 一旦它找到了`mystuff`，就轮到处理句点`.`(period)这个操作符，而且开始查看`mystuff`内部的一些变量了。由于`mystuff`是一个列表，Python 知道它支持一些函数。
> 3. 接下来轮到了处理`append`。Python会将“append”和`mystuff`支持的所有函数的名称一一对比，如果确实其中有一个叫`append`的函数，那么 Python 就会去使用这个函数。
> 4. 接下来 Python 看到了括号`(`(parenthesis) 并且意识到, “噢，原来这应该是一个函数”，到了这里，它就正常会调用这个函数了，不过这里的函数还要多一个参数才行。
> 5. 这个额外的参数其实是……`mystuff`!我知道，很奇怪是不是？不过这就是 Python 的工作原理，所以还是记住这一点，就当它是正常的好了。真正发生的事情其实是`append(mystuff, 'hello')`，不过你看到的只是`mystuff.append('hello')`。

大部分时候你不需要知道这些细节，不过如果你看到一个像这样的 Python 错误信息的时候，上面的细节就对你有用了：

```
$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> class Thing(object):
...     def test(hi):
...             print "hi"
...
>>> a = Thing()
>>> a.test("hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: test() takes exactly 1 argument (2 given)
>>>
```

就是这个吗？嗯，这个是我在Python命令行下展示给你的一点魔法。你还没有见过 `class` 不过后面很快就要碰到了。现在你看到 Python 说 `test() takes exactly 1 argument (2 given)` (test() 只可以接受1个参数，实际上给了两个)。它意味着 python 把 `a.test("hello")` 改成了 `test(a,"hello")` ，而有人弄错了，没有为它添加 `a` 这个参数。

一下子要消化这么多可能有点难度，我们将做几个练习，让你头脑中有一个深刻的印象。下面的练习将字符串和列表混在一起，看看你能不能在里边找出点乐子来：

```
ten_things = "Apples Oranges Crows Telephone Light Sugar"

print "Wait there are not 10 things in that list. Let's fix that."

stuff = ten_things.split(' ')
more_stuff = ["Day", "Night", "Song", "Frisbee", "Corn", "Banana", "Girl", "Boy"]

while len(stuff) != 10:
    next_one = more_stuff.pop()
    print "Adding: ", next_one
    stuff.append(next_one)
    print "There are %d items now." % len(stuff)

print "There we go: ", stuff

print "Let's do some things with stuff."

print stuff[1]
print stuff[-1] # whoa! fancy
print stuff.pop()
print ' '.join(stuff) # what? cool!
print '#'.join(stuff[3:5]) # super stellar!
```

你看到的结果

```
$ python ex38.py
Wait there are not 10 things in that list. Let's fix that.
Adding:  Boy
There are 7 items now.
Adding:  Girl
There are 8 items now.
Adding:  Banana
There are 9 items now.
Adding:  Corn
There are 10 items now.
There we go:  ['Apples', 'Oranges', 'Crows', 'Telephone', 'Light', 'Sugar', 'Boy', 'Gi
rl', 'Banana', 'Corn']
Let's do some things with stuff.
Oranges
Corn
Corn
Apples Oranges Crows Telephone Light Sugar Boy Girl Banana
Telephone#Light
```

列表能实现什么

假设你打算创建一个基于钓鱼的电脑游戏。如果你不知道什么是钓鱼，花点时间在网上找到相关资料看一看。要做到这些，你需要里了解一些关于“扑克牌”的概念，并将它们变成你的Python程序。你必须用python写出知道如何玩一副虚拟扑克牌的代码，这样人们就能把你的游戏当成真实的游戏来玩，即使它不是真的。你需要的是一副“扑克牌”结构，而程序员就称之为“数据结构”。

什么是数据结构呢？如果你仔细想想，数据结构其实就是一种正式的构造（组织）一些数据（事实）的方法。真的就是这么简单，尽管一些数据结构十分错综复杂，它们也仅仅是在程序中存储数据的方式而已，这样你就能用不同的方式访问它们。

我将在下一节练习中更深入的讲解这些，但是列表是程序员常用的数据结构之一。它们只是数据的有序列表，你可以通过线性索引来存储或访问它们。什么？记住我说过的话，只是因为一个程序员说过“列表是一个列表”并不意味着它比真实世界中的列表复杂多少。 让我们以扑克牌为例：

> 1. 你有一堆有值的卡片。
> 2. 这些卡片式一堆，一列，或者可以从头到尾排列这些卡片。
> 3. 你可以随机的从顶部、中部或者底部取出卡片。
> 4. 如果你想找到某张特殊的卡片, 你必须一张一张的检索这些卡片。

让我们看看我说过什么：

“一个有序的列表”是的，扑克牌有第一张和最后一张"有一些你要存储的事物"是的，卡片就是我要存储的东西"可以随机访问"是的，在这副牌中我可以任意取出一张."是线性的"是的，如果我想找到特定的一张牌，我必须从头开始按顺序查找。"有索引的"差不多, 如果我让你找到一副扑克牌的第19张，你就必须按照顺序去数，直到你找到这一张。在python的列表中，计算机可以直接跳到你给出的索引处。这就是一个列表做的所有的事情，这么解释应该给了你一个在编程中找到列表概念的方法吧。编程中的每一个概念通常都有与真实世界相关联，至少在真实世界是有用的。如果你能找出虚拟的概念在真实世界中对应的概念，那你也可以通过这些找到数据结构到底是做什么的。

什么情况可以使用列表

当你有一些有东西能匹配列表数据结构的有用特性时，使用列表:

> 1. 如果你需要维护一组次序，记住，这是把次序编成列表，而不是给次序排序，列表不会帮你排序。
> 2. 如果你需要根据一个随机数访问列表内容，记住，这时候基数从0开始。
> 3. 如果你需要从头到尾操作列表内容，记住，这就是for循环。

附加题

> 1. 将每一个被调用的函数以上述的方式翻译成 Python 实际执行的动作。比如 `more_stuff.pop()` 是 `pop(more_stuff)`.
> 2. 将这两种方式翻译为自然语言。
> 3. 上网阅读一些关于"面向对象编程(Object Oriented Programming)"的资料。晕了吧？嗯，我以前也是。别担心。你将从这本书学到足够用的关于面向对象编程的基础知识，而以后你还可以慢慢学到更多。
> 4. 查一下 Python中的 "class" 是什么东西。不要阅读关于其他语言的 "class" 的用法，这会让你更糊涂。
> 5. 如果你不知道我讲的是些什么东西，别担心。程序员为了显得自己聪明，于是就发明了 Opject Oriented Programming，简称为 OOP，然后他们就开始滥用这个东西了。如果你觉得这东西太难，你可以开始学一下 "函数编程(functional programming)"。
> 6. 找出10种可以放在列表中的例子，并用它们写一些脚本。

常见问题

Q：你没有说说不要用while循环？

> 是的，请记住，当需要的时候，你可以打破规则，只有蠢人才会一直一味的遵从规则。

Q: `join(' ', stuff)` 为什么没有生效？

> 文档中关于join的内容，写的没有意义，`join` 是使用一个字符串将列表内容链接起来的一个方法，你可以试试这么写 `' '.join(stuff).`.

Q: 为什么你用了一个while循环？

> 试着用for循环改写一下，看看哪个更简单

Q: `stuff[3:5]` 实现了什么功能？

> 这句代码从 `stuff` 中获取了一个子集，包含stuff的第3和第4个元素，没有包含第5个元素。它和 `range(3,5)` 的工作原理相近。

练习39.字典,可爱的字典

接下来我要教你另外一种让你伤脑筋的容器型数据结构，因为一旦你学会这种容器，你将拥有超酷的能力。这是最有用的容器：字典(dictionary)。

Python 将这种数据类型叫做 “dict”，有的语言里它的名称是 “hash”。这两种名字我都会用到，不过这并不重要，重要的是它们和列表的区别。你看，针对列表你可以做这样的事情：

```
>>> things = ['a', 'b', 'c', 'd']
>>> print things[1]
b
>>> things[1] = 'z'
>>> print things[1]
z
>>> things
['a', 'z', 'c', 'd']
```

你可以使用数字作为列表的索引，也就是你可以通过数字找到列表中的元素。你现在应该了解列表的这些特性，而你也应了解，你也只能通过数字来获取列表中的元素。

而 dict 所作的，是让你可以通过任何东西找到元素，不只是数字。是的，字典可以将一个物件和另外一个东西关联，不管它们的类型是什么，我们来看看：

```
>>> stuff = {'name': 'Zed', 'age': 39, 'height': 6 * 12 + 2}
>>> print stuff['name']
Zed
>>> print stuff['age']
39
>>> print stuff['height']
74
>>> stuff['city'] = "San Francisco"
>>> print stuff['city']
San Francisco
```

你将看到除了通过数字以外，我们还可以用字符串来从字典中获取 stuff ，我们还可以用字符串来往字典中添加元素。当然它支持的不只有字符串，我们还可以做这样的事情：

```
>>> stuff[1] = "Wow"
>>> stuff[2] = "Neato"
>>> print stuff[1]
Wow
>>> print stuff[2]
Neato
>>> stuff
{'city': 'San Francisco', 2: 'Neato', 'name': 'Zed', 1: 'Wow', 'age': 39, 'height': 74
}
```

在这段代码中，我使用了数字，当我打印stuff的时候，你可以看到，不止有数字还有字符串作为字典的key。事实上，我可以使用任何东西，这么说并不准确，不过你先这么理解就行了。

当然了，一个只能放东西进去的字典是没啥意思的，所以我们还要有删除的方法，也就是使用 `del` 这个关键字：

```
>>> del stuff['city']
>>> del stuff[1]
>>> del stuff[2]
>>> stuff
{'name': 'Zed', 'age': 36, 'height': 74}
```

一个字典实例

接下来我们要做一个练习，你必须非常仔细，我要求你将这个练习写下来，然后试着弄懂它做了些什么。注意一下这个例子中是如何对应这些州和它们的缩写，以及这些缩写对应的州里的城市。 记住, "映射" 是字典中的关键概念.

你看到的结果

```
# create a mapping of state to abbreviation
states = {
    'Oregon': 'OR',
    'Florida': 'FL',
    'California': 'CA',
    'New York': 'NY',
    'Michigan': 'MI'
}

# create a basic set of states and some cities in them
cities = {
    'CA': 'San Francisco',
    'MI': 'Detroit',
    'FL': 'Jacksonville'
}

# add some more cities
cities['NY'] = 'New York'
cities['OR'] = 'Portland'

# print out some cities
print '-' * 10
print "NY State has: ", cities['NY']
print "OR State has: ", cities['OR']

# print some states
print '-' * 10
print "Michigan's abbreviation is: ", states['Michigan']
print "Florida's abbreviation is: ", states['Florida']

# do it by using the state then cities dict
print '-' * 10
print "Michigan has: ", cities[states['Michigan']]
print "Florida has: ", cities[states['Florida']]

# print every state abbreviation
print '-' * 10
for state, abbrev in states.items():
    print "%s is abbreviated %s" % (state, abbrev)

# print every city in state
print '-' * 10
for abbrev, city in cities.items():
    print "%s has the city %s" % (abbrev, city)

# now do both at the same time
print '-' * 10
for state, abbrev in states.items():
    print "%s state is abbreviated %s and has city %s" % (
        state, abbrev, cities[abbrev])

print '-' * 10
# safely get a abbreviation by state that might not be there
state = states.get('Texas')

if not state:
    print "Sorry, no Texas."

# get a city with a default value
city = cities.get('TX', 'Does Not Exist')
print "The city for the state 'TX' is: %s" % city
```

你看到的结果

```
$ python ex39.py
----------
NY State has:  New York
OR State has:  Portland
----------
Michigan's abbreviation is:  MI
Florida's abbreviation is:  FL
----------
Michigan has:  Detroit
Florida has:  Jacksonville
----------
California is abbreviated CA
Michigan is abbreviated MI
New York is abbreviated NY
Florida is abbreviated FL
Oregon is abbreviated OR
----------
FL has the city Jacksonville
CA has the city San Francisco
MI has the city Detroit
OR has the city Portland
NY has the city New York
----------
California state is abbreviated CA and has city San Francisco
Michigan state is abbreviated MI and has city Detroit
New York state is abbreviated NY and has city New York
Florida state is abbreviated FL and has city Jacksonville
Oregon state is abbreviated OR and has city Portland
----------
Sorry, no Texas.
The city for the state 'TX' is: Does Not Exist
```

字典能做什么

字典是另一个数据结构的例子，和列表一样，是编程中最常用的数据结构之一。 字典是用来做映射或者存储你需要的键值对，这样当你需要的时候，你可以通过key来获取它的值。同样，程序员不会使用一个像“字典”这样的术语，来称呼那些不能像一个写满词汇的真实字典正常使用的事物，所以我们只要把它当做真实世界中的字典来用就好。

假如你想知道这个单词"Honorificabilitudinitatibus"的意思。你可以很简单的把它复制粘贴放进任何一个搜索引擎中找到答案。我们真的可以说一个搜索引擎就像一个巨大的超级复杂版本的《牛津英语词典》(OED).在搜索引擎出现之前，你可能会这样做：

> 1. 走进图书馆，找到一本字典，我们称这本字典为OED
> 2. 你知道单词"honorificabilitudinitatibus" 以字母 'H'开头，所以你查看字典的小标签，找到以 'H' 开头的部分.
> 3. 然后你会浏览书页，直到找到"hon"开头的地方。
> 4. 然后你再翻过一些书页，直到找到 "honorificabilitudinitatibus" 或者找到以 "hp" 开头的单词，发现这个词不在我们的字典中。
> 5. 当你找到这个条目，你就可以仔细阅读并弄明白它的意思。

这个过程跟我们在程序中使用字典的是相似的，你会映射（"mapping"）找到这个单词"honorificabilitudinitatibus"的定义。Python中的字典就跟真实世界中的这本牛津词典（OED）差不多。

定义自己的字典类

这节练习的最后一段代码给你演示了如何使用你刚学会的list来创建一个字典数据结构。这段代码可能有些难以理解，所以如果你要花费你很长的时间去弄明白代码额含义也不要担心。代码中会有一些新的知识点，它确实有些复杂，还有一些事情需要你上网查找

为了使用Python中的 `dict` 保存数据，我打算把我的数据结构叫做 `hashmap`,这是字典数据结构的另一个名字。你要把下面的代码输入一个叫做 `hashmap.py` 的文件，这样我们就可以在另一个文件 `ex39_test.py` 中执行它。

```
def new(num_buckets=256):
    """Initializes a Map with the given number of buckets."""
    aMap = []
    for i in range(0, num_buckets):
        aMap.append([])
    return aMap

def hash_key(aMap, key):
    """Given a key this will create a number and then convert it to
    an index for the aMap's buckets."""
    return hash(key) % len(aMap)

def get_bucket(aMap, key):
    """Given a key, find the bucket where it would go."""
    bucket_id = hash_key(aMap, key)
    return aMap[bucket_id]

def get_slot(aMap, key, default=None):
    """
    Returns the index, key, and value of a slot found in a bucket.
    Returns -1, key, and default (None if not set) when not found.
    """
    bucket = get_bucket(aMap, key)

    for i, kv in enumerate(bucket):
        k, v = kv
        if key == k:
            return i, k, v

    return -1, key, default

def get(aMap, key, default=None):
    """Gets the value in a bucket for the given key, or the default."""
    i, k, v = get_slot(aMap, key, default=default)
    return v

def set(aMap, key, value):
    """Sets the key to the value, replacing any existing value."""
    bucket = get_bucket(aMap, key)
    i, k, v = get_slot(aMap, key)

    if i >= 0:
        # the key exists, replace it
        bucket[i] = (key, value)
    else:
        # the key does not, append to create it
        bucket.append((key, value))

def delete(aMap, key):
    """Deletes the given key from the Map."""
    bucket = get_bucket(aMap, key)

    for i in xrange(len(bucket)):
        k, v = bucket[i]
        if key == k:
            del bucket[i]
            break

def list(aMap):
    """Prints out what's in the Map."""
    for bucket in aMap:
        if bucket:
            for k, v in bucket:
                print k, v
```

上面的代码创建了一个叫做 `hashmap` 的模块，你需要把这个模块import到文件 `ex39_test.py` 中，并让这个文件运行起来：

```
import hashmap

# create a mapping of state to abbreviation
states = hashmap.new()
hashmap.set(states, 'Oregon', 'OR')
hashmap.set(states, 'Florida', 'FL')
hashmap.set(states, 'California', 'CA')
hashmap.set(states, 'New York', 'NY')
hashmap.set(states, 'Michigan', 'MI')

# create a basic set of states and some cities in them
cities = hashmap.new()
hashmap.set(cities, 'CA', 'San Francisco')
hashmap.set(cities, 'MI', 'Detroit')
hashmap.set(cities, 'FL', 'Jacksonville')

# add some more cities
hashmap.set(cities, 'NY', 'New York')
hashmap.set(cities, 'OR', 'Portland')

# print out some cities
print '-' * 10
print "NY State has: %s" % hashmap.get(cities, 'NY')
print "OR State has: %s" % hashmap.get(cities, 'OR')

# print some states
print '-' * 10
print "Michigan's abbreviation is: %s" % hashmap.get(states, 'Michigan')
print "Florida's abbreviation is: %s" % hashmap.get(states, 'Florida')

# do it by using the state then cities dict
print '-' * 10
print "Michigan has: %s" % hashmap.get(cities, hashmap.get(states, 'Michigan'))
print "Florida has: %s" % hashmap.get(cities, hashmap.get(states, 'Florida'))

# print every state abbreviation
print '-' * 10
hashmap.list(states)

# print every city in state
print '-' * 10
hashmap.list(cities)

print '-' * 10
state = hashmap.get(states, 'Texas')

if not state:
  print "Sorry, no Texas."

# default values using ||= with the nil result
# can you do this on one line?
city = hashmap.get(cities, 'TX', 'Does Not Exist')
print "The city for the state 'TX' is: %s" % city
```

你应该发现这段代码跟我们在这节开始时写的代码几乎相同，除了它使用了你新实现的HashMap。通读代码并且确信你明白这段代码中的每一行是如何与 `ex39.py` 中的代码实现相同的功能。

代码分析

这个 `hashmap` 只不过是"拥有键值对的有插槽的列表"，用几分钟时间分析一下我说的意思：

"一个列表"在 `hashmap` 函数中，我创建了一个列表变量`aMap`，并且用其他的列表填充了这个变量。"有插槽的列表"最开始这个列表是空的,当我给这个数据结构添加键值对之后，它就会填充一些插槽或者其他的东西"拥有键值对"表示这个列表中的每个插槽都包含一个`(key, value)`这样的元素或者数据对。

如果我的这个描述仍旧没让你弄明白是什么意思，花点时间在纸上画一画它们，直到你搞明白为止。实际上，手动在纸上运算是让你弄明白它们的好办法。

你现在知道数据是如何被组织起来的，你还需要知道它每个操作的算法。算法指的是你做什么事情的步骤。它是是数据结构运行起来的代码。我们接下来要逐个分析下代码中用到的操作，下面是在 `hashmap` 算法中一个通用的模式：

> 1. 把一个关键字转换成整数使用哈希函数: `hash_key`.
> 2. Convert this hash to a bucket number using a `%` (模除) 操作.
> 3. Get this bucket from the `aMap` list of buckets, and then traverse it to find the slot that contains the key we want.

操作`set` 实现以下功能,如果key值存在，则替换原有的值，不存在则创建一个新值。

下面我们逐个函数分析一下hashmap的代码，让你明白它是如何工作的。 跟我一起分析并确保你明白每一行代码的意思。给每一行加上注释，确保你明白他们是做什么的。就是如此简单，我建议你对下面提到的每一行代码都花点时间在Python shell或者纸上多练习练习:

new首先，我以创建一个函数来生成一个hashmap开始，也被称为初始化。 我先创建一个包含列表的变量，叫做`aMap`,然后把列表`num_buckets`放进去， `num_buckets`用来存放我给hashmap设置的内容。 后面我会在另一个函数中使用 `len(aMap)` 来查找一共有多少个buckets。确信你明白我说的。

hash_key这个看似简单的函数是一个dict如何工作的核心。它是用Python内建的哈希函数将字符串转换为数字。Python为自己的字典数据结构使用此功能，而我只是复用它. 你应该启动一个Python控制台，看看它是如何工作的. 当我拿到key对应的数字的时候, 我使用 `%` (模除)操作和 `len(aMap)` 来获得一个放置这个key的位置。你应该知道，`%` (模除) 操作将会返回除法操作的余数。我也可以使用这个方法来限制大数，将其固为较小的一组数字。如果你不知道我在说什么，使用Python解析器研究一下。

get_bucket这个函数使用`hash_key`来找到一个key所在的“bucket”。当我在`hash_key`函数中进行`%len(aMap)`操作的时候，我知道无论我获得哪一个 `bucket_id` 都会填充进 `aMap` 列表中. 使用 `bucket_id` 可以找到一个key所在的“bucket” 。

get_slot这个函数使用`get_bucket`来获得一个key所在的“bucket”， 它通过查找bucket中的每一个元素来找到对应的key。找到对应的key之后，它会返回这样一个元组`(i, k, v)`，i表示的是key的索引值，k就是key本身，v是key对应的值。你现在已经了解了足够字典数据结构运行的原理。它通过keys、哈希值和模量找到一个bucket，然后搜索这个bucket，找到对应的条目。它有效的减少了搜索次数。

get这是一个人们需要`hashmap`的最方便的函数。它使用`get_slot`来获得 元组`(i, k, v)`但是只返回`v`. 确定你明白这些默认变量是如何运行的，以及`get_slot`中`(i, k, v)`分派给`i`, `k`, `v`的变量是如何获得的。

set设置一个`key/value`键值对，并将其追加到字典中，保证以后再用到的时候可以获取的到。但是，我希望我的`hashmap`每个key值存储一次。为了做到这点， 首先，我要找到这个key是不是已经存在，如果存在，我会替换它原来的值，如果不存在，则会追加进来。这样做会比简单的追加慢一些，但是更满足`hashmap`使用者的期望。如果你允许一个key有多个value，你需要使用 `get` 方法查阅所有的“bucket”并返回一个所有value的列表。这是平衡设计的一个很好的例子。现在的版本是你可以更快速的 `get`, 但是会慢一些 `set`.

delete删除一个key, 找到key对应的 bucket，并将其从列表中删除。因为我选择一个key只能对应一个value，当我找到一个相应的key的时候，我就可以停止继续查找和删除。如果我选择允许一个key可以对应多个value的话，我的删除操作也会慢一些，因为我要找到所有key对应的value，并将其删除。

list最后的功能仅仅是一个小小的调试功能，它能打印出`hashmap` 中的所有东西，并且能帮助你理解字典的细微之处。

在所有的函数之后，我有一点点的测试代码，可以确保他们正常工作。

三级列表

正如我在讨论中提到的， 由于我选择 `set` 来重写 (替换) 原有的keys，这会让`set`执行的慢一些，但是这决定能让其他的一些函数快一些。如果我想要一个`hashmap`允许一个key对应多个value，但又要求所有函数仍然执行的很快，怎么办

我要做的就是让每个“bucket”中的插槽的值都是一个列表。这意味着我要给字典再增加第三级列表。这种`hashmap`仍然满足字典的定义。我说过， "每一个key可以有对个value"的另一种说法就是"每一个key有一个value的列表"。

你应该看到的结果（again）

```
$ python ex39_test.py
Traceback (most recent call last):
  File "ex39_test.py", line 1, in <module>
    import hashmap
ImportError: No module named hashmap
```

什么时候选择字典，什么时候选择列表

如同我在练习38中提到的，列出有特定的特性，帮助你控制和组织需要放在表结构的东西。字典也一样，但是 `dict` 的特性是与列表不同的，因为他们是用键值对映射来工作的。当遇到下面情况的时候，可以使用字典：

> 1. 你要检索的东西是以一些标识为基础的，比如名字、地址或其他一切可以作为key的东西。
> 2. 你不需要这些东西是有序的。词典通常不会有序的概念，所以你必须使用一个列表。
> 3. 你想要通过key增删一个元素。

也就是说，如果你要用一个非数字的key，使用 `dict` ，如果你需要有序的东西，使用 `list` .

附加题

> 1. 用自己国家的州和城市做一些类似的映射关系
> 2. 在 Python 文档中找到 dictionary 的相关的内容，学着对 dict 做更多的操作。
> 3. 找出一些 dict 无法做到的事情。例如比较重要的一个就是 dict 的内容是无序的，你可以检查一下看看是否真是这样。
> 4. 阅读python的关于断言的功能，然后修改hashmap的代码，给每一个测试增加一些断言相关的代码，替换原来的打印代码。比如，你可以断言第一个操作返回"Flamenco Sketches" 而不是直接打印出"Flamenco Sketches" 。
> 5. 有没有注意到 `list` 方法没有按照你增加元素的顺序把它们列出来？这是字典不维持秩序的一个例子，如果你将代码进行分析，你就会明白为什么。
> 6. 像写 `list` 方法一样写一个 `dump` 方法，实现备份字典内所有内容的功能
> 7. 确定你知道 `hash` 在代码中实现了什么功能，它是一个特殊的函数，能将字符串转化为整数。在网上找到相关文档，了解在程序设计中，什么是哈希函数。

常见问题

Q：字典和列表的区别？

> list是一个有序的项目列表，dict是匹配一些项目（称为“键”）和其他项目（称为“值”）。

Q: 我怎样使用字典？

> 当你需要通过一个值来访问另一个值的时候，使用字典。实际上，你可以把字典称作“对照表”

Q: 我怎样使用列表？

> 对任何需要有序的事情集合使用列表，你只需要通过他们的数值索引来访问它们。

Q:加入我需要一个字典，但是又需要是有序的，怎么办？

> 看看python中`collections.OrderedDict`这个数据结构。网上搜索相关的文档。

练习40.模块, 类和对象

Python 是一门“面向对象编程语言”。这意味着在Python中有一个叫做类的概念，你能通过类用一种特殊的方式构建你的软件。使用类的概念，能给你的程序增添一致性，这样你可以用一种很轻松方便的方式调用他们。至少，这是面向对象的理论。

我现在要通过使用你已经知道的字典和模块等来教你开始学习面向对象编程、类和对象。我的问题是面向对象编程（OOP）只是普通的怪异。你要为之而奋斗，努力尝试理解我讲的内容，编写代码，在下一个练习中，我会更深入的讲解。

我们要开始了。

模块就像字典

你知道字典是如何被创建以及使用的，它用来将一个事物对应到另一个事物。意思就是说如果你有一个字典，字典中包括一个key "apple"，那么你就可以实现：

```
mystuff = {'apple': "I AM APPLES!"}
print mystuff['apple']
```

记住“从X获取Y”的说法，然后想一想模块（module）。你之前已经写过一些模块，你应该知道它们：

> 1. 包含一些函数和变量的python文件
> 2. 你可以导入这个文件
> 3. 你可以用 `.` 操作符访问这个模块的函数和变量

想象一下我有一个叫做 `mystuff.py` 的模块，在这个模块中有一个叫做 `apple` 的函数，下面是 `mystuff.py` 模块的代码：

```
# this goes in mystuff.py
def apple():
    print "I AM APPLES!"
```

当我写完以上代码，我就可以通过import来调用 `mystuff` 模块，并访问 `apple` 函数：

```
import mystuff
mystuff.apple()
```

我还可以增加一个叫做 `tangerine` 的变量:

```
def apple():
    print "I AM APPLES!"

# this is just a variable
tangerine = "Living reflection of a dream"
```

可以用相同的方法来访问：

```
import mystuff

mystuff.apple()
print mystuff.tangerine
```

返回再看字典，你应该发现模块的使用跟字典是相似的，只不过语法不同，我们比较一下：

```
mystuff['apple'] # get apple from dict
mystuff.apple() # get apple from the module
mystuff.tangerine # same thing, it's just a variable
```

也就是说我们在Python中有一套通用的模式：

> 1. 有一个key=value模式的容器
> 2. 通过key从容器中获取数据

在字典中，key是一个字符串，写法为 `[key]` ,在模块中，key是一个标识符，写法为 `.key` ，在其余的地方，他们几乎是一样的。

类就像模块

你可以认为一个模块就是一个可以存放Python代码的特殊字典，这样你就可以通过 `.` 操作符访问它。Python还有一个另外的结构提供相似的功能，就是类（class）。类的作用是组织一系列的函数和数据并将它们放在一个容器里，这样你可以通过 `.` 操作符访问到它们。

如果我想创建一个类似 `mystuff` 的类，我需要这样做：

```
class MyStuff(object):

    def __init__(self):
        self.tangerine = "And now a thousand years between"

    def apple(self):
        print "I AM CLASSY APPLES!"
```

这相比于模块复杂一些，对比之下肯定会有不同，但是你应该能返现这个类就像写一个包含 `apple()` 方法的“迷你” `mystuff` 模块一样.让你困惑的地方可能是 `__init__()` 这个方法以及使用 `self.tangerine` 给 `tangerine` 赋值。

这正是为什么要使用类而不是仅有模块的原因：你可以使用 `Mystuff` 这个类，还可以用它来创建更多个 `Mystuff` ，而他们之间也不会互相冲突。当你导入一个模块时，你的整个项目也就只有一个这个模块。

在你理解这些之前，你需要明白什么是“对象”，以及如何像使用 `Mystuff.py` 模块一样使用 `Mystuff` 这个类。

对象就像导入

如果一个类就像一个迷你模块，那么类也会有一个类似 `import` 的概念，这个概念被称为实例化，这只是对创建一种更虚幻更聪明的叫法。当一个类被实例化，你就得到一个类的对象。

实例化类的方法就是像调用函数一样调用这个类：

```
thing = MyStuff()
thing.apple()
print thing.tangerine
```

第一行就是实例化的操作，这个操作多像是在调用一个函数啊。当然了，python在幕后帮你做了一系列的事情，我带你来看看具体的调用步骤：

> 1. python 查找 `MyStuff()` 并确认它是你已经定义过的类
> 2. python创建一个空的对象，该对象拥有你在类中用 `def` 创建的所有函数
> 3. python看你是否创建了 `__init__` 函数，如果有，调用该方法初始化你新创建的空对象
> 4. 在 `MyStuff` 中， `__init__` 方法有一个额外的变量 `self` ，这是python为我创建的一个空的对象，我可以在其上设置变量。
> 5. 然后，我给 `self.tangerine` 设置一首歌词，然后初始化这个对象
> 6. python可以使用这个新创建好的对象，并将其分配给我可以使用的变量 `thing` 。

这就是当你调用一个类的时候，python做的事情。记住，这并不是把类给你，而是把类作为蓝本来创建这种类型东西的副本。

我给了你一个类的简单工作原理，这样你就可以基于你所了解的模块，建立你对类的理解。事实上，在这一点上，类和对象与模块是有区别的：

> - 类是用来创建迷你模块的蓝本或定义
> - 实例化是如何创建这些小模块，并在同一时间将其导入。实例化仅仅是指通过类创建一个对象。
> - 由此产生的迷你模块被称为对象，你可以将其分配给一个变量，让它开始运行

在这一点上，对象和模块的表现不同，这只是提供一种让你理解类和对象的方法。

从一个事物中获取事物

现在我提供给你3中方法从一个事物中获取另一个：

```
# dict style
mystuff['apples']

# module style
mystuff.apples()
print mystuff.tangerine

# class style
thing = MyStuff()
thing.apples()
print thing.tangerine
```

类的举例

你应该可以看到在这三个key=value的容器类型有一定的相似性，你也可能有一堆问题.记住这些问题，下一节练习会训练你关于"面向对象的词汇"。这节练习中，我只想让你输入这些代码并保证它能正常运行，这样能让你再继续下一节练习之前获得一些经验。

```
class Song(object):

    def __init__(self, lyrics):
        self.lyrics = lyrics

    def sing_me_a_song(self):
        for line in self.lyrics:
            print line

happy_bday = Song(["Happy birthday to you",
                   "I don't want to get sued",
                   "So I'll stop right there"])

bulls_on_parade = Song(["They rally around tha family",
                        "With pockets full of shells"])

happy_bday.sing_me_a_song()

bulls_on_parade.sing_me_a_song()
```

你应该看到的内容

```
$ python ex40.py
Happy birthday to you
I don't want to get sued
So I'll stop right there
They rally around tha family
With pockets full of shells
```

附加题

> 1. 用本节学到的内容多写几首歌，确定你明白你把一个字符串的列表作为歌词传递进去。
> 2. 把歌词放进一个单独的变量，再把这个变量传递给类
> 3. 看看你能不能让这个类做更多的事情，如果没什么想法也没关系，试试看，会有什么
> 4. 在网上搜索一些“面向对象编程”的资料，尝试让你的大脑填满这些资料。如果这些资料对你没有用，也没关系，这些东西有一半对我来说也是没有意义的。

常见问题

Q：为什么我在类里创建 `__init__` 或其他函数的时候需要使用 `self` ？

> 如果你不实用`self` ，像这种代码`cheese = 'Frank'`就是不明确的. 代码并不清楚你指的是实例的`cheese`属性，还是一个叫做`cheese`d的全局变量。如果你使用`self.cheese='Frank'`代码就会十分清楚你指的就是实例中的属性`self.cheese`.

练习41.学会说面向对象

在这个练习中，我要教你如何说“面向对象”，我要给你一些你需要知道定义的词。然后我会给你一组你必须了解的句子，最后我会给你一大堆练习，你必须完成这练习题，将我给你的句子转化成自己的词汇。

单词解释

class(类)：告诉python去创建一个新类型。object(对象)：有两种意思，事物的基本类型，或者事物的实例化。instance(实例)：你通过python创建一个类所获得的。def：用来在类中定义一个函数。self：在一个类包含的函数中，self是一个用来访问实例或对象的变量。inheritance：概念，表示一个类可以继承另一个类的特征,就像你和你的父母。composition：概念，表示一个类可以包含其他类,就像汽车轮子。 attribute：类所拥有的特性，通常是变量。is-a：惯用语，表示一个东西继承自另一个东西(a),像在“鲑鱼”是“鱼”。 has-a：惯用语，表示由其他事情或有一个特征(a),如“鲑鱼有嘴。”

花一些时间制作一些卡片用来记忆这些术语。像往常一样，直到你完成这个练习后，这都不会有太多的意义，但是首先你需要知道的基本词汇。

短语解释

接下来,在左边有一个Python代码片段列表，右面是他们的解释 `class X(Y)` ：创建一个叫X的类，并继承Y。 `class X(object): def __init__(self, J)` ：类X有一个 `__init__` 方法，该方法有self和J两个参数。 `class X(object): def M(self, J)` ：类X有一个叫M的函数，该函数有self和J两个参数。 `foo = X()` ：给foo赋值为类X的一个实例。 `foo.M(J)` ：从foo里调用M函数，传递的参数为self和J。 `foo.K = Q` ：从foo里调用K属性，并将其设置为Q。

你可以把上面看到的所有的X, Y, M, J, K, Q, 以及 foo 看做空白的坑，比如，我还可以这么写：

> 1. 创建一个叫？？的类继承Y
> 2. 类？？有一个 `__init__` 方法，该方法有self和？？两个参数。
> 3. 类？？有一个叫？？的函数，该函数有self和？？两个参数。
> 4. 给foo赋值为类？？的一个实例。
> 5. 从foo里调用？？函数，传递的参数为self和？？。
> 6. 从foo里调用？？属性，并将其设置为？？。

同样的，把这些写到卡片上，牢牢记住它们。卡片的前面写上python的小段代码，背面写上它们的解释，你要做到每当你看到正面的代码段的时候，能立即说出后面的解释。

组合练习

最后给你准备的是将单词和短语结合起来练习。我希望你能做到下面的要求：

> 1. 准备好短语的卡片，并拼命的练习
> 2. 翻转卡片，阅读这些解释语句，挑选出语句中包含单词练习中单词的卡片
> 3. 通过这些语句拼命练习这些单词
> 4. 坚持练习，直到你厌烦了，休息一下，然后继续练习

阅读测试

下面有一个python脚本，这个脚本会以无尽模式训练你，检验你所掌握的这些单词。这是一个很简单的脚本，它实现的功能是使用了一个叫做`urllib`的类库来下载我提供的单词列表。下面就是这个脚本，你需要正确的输入并命名为`oop_test.py`：

```
import random
from urllib import urlopen
import sys

WORD_URL = "http://learncodethehardway.org/words.txt"
WORDS = []

PHRASES = {
    "class %%%(%%%):":
      "Make a class named %%% that is-a %%%.",
    "class %%%(object):\n\tdef __init__(self, ***)" :
      "class %%% has-a __init__ that takes self and *** parameters.",
    "class %%%(object):\n\tdef ***(self, @@@)":
      "class %%% has-a function named *** that takes self and @@@ parameters.",
    "*** = %%%()":
      "Set *** to an instance of class %%%.",
    "***.***(@@@)":
      "From *** get the *** function, and call it with parameters self, @@@.",
    "***.*** = '***'":
      "From *** get the *** attribute and set it to '***'."
}

# do they want to drill phrases first
if len(sys.argv) == 2 and sys.argv[1] == "english":
    PHRASE_FIRST = True
else:
    PHRASE_FIRST = False

# load up the words from the website
for word in urlopen(WORD_URL).readlines():
    WORDS.append(word.strip())

def convert(snippet, phrase):
    class_names = [w.capitalize() for w in
                   random.sample(WORDS, snippet.count("%%%"))]
    other_names = random.sample(WORDS, snippet.count("***"))
    results = []
    param_names = []

    for i in range(0, snippet.count("@@@")):
        param_count = random.randint(1,3)
        param_names.append(', '.join(random.sample(WORDS, param_count)))

    for sentence in snippet, phrase:
        result = sentence[:]
```

```
            # fake class names
            for word in class_names:
                result = result.replace("%%%", word, 1)

            # fake other names
            for word in other_names:
                result = result.replace("***", word, 1)

            # fake parameter lists
            for word in param_names:
                result = result.replace("@@@", word, 1)

            results.append(result)

        return results

# keep going until they hit CTRL-D
try:
    while True:
        snippets = PHRASES.keys()
        random.shuffle(snippets)

        for snippet in snippets:
            phrase = PHRASES[snippet]
            question, answer = convert(snippet, phrase)
            if PHRASE_FIRST:
                question, answer = answer, question

            print question

            raw_input("> ")
            print "ANSWER:  %s\n\n" % answer
except EOFError:
    print "\nBye"
```

运行这个脚本，尝试将这些“面向对象的短语”翻译成自己的语言。你应该能看到字典 `PHRASES` 中包含了刚才练习的所有的短语。

练习将英语转换为代码

接下来，你可以使用"english"选项来执行脚本，这样你可以反过来练习：

```
$ python oop_test.py english
```

记住这些短语使用的是无意义的词汇。学习阅读代码的一部分是停止纠结这些用于变量和类的名字的真实意义。人们常常会在读到一个像“cork”的词时突然迷糊,因为这个词会混淆他们的意义。在这个例子中，"Cork"只是用来作为一个类的名字。不要给它任何意义的解释。

阅读更多的代码

你现在需要继续阅读更多的代码，阅读你找到的代码中这些你刚学过的短语表达。你需要找出文件中所有的类，然后执行以下步骤：

> 1. 给出每一个类的名字，并说出这些类继承哪些类
> 2. 列出每个类所包含的函数，以及函数需要的参数
> 3. 列出类所有的属性
> 4. 对每个属性，给出属性的类型

这个练习的目的是通过阅读真实的代码，学习你刚才学到的短语是如何使用的。如果你练习的足够所，你应该能看到这些模式在代码中向你大声呼喊，然而在这之前，他们是你所不知道的，只是模糊的空白而已。

常见问题

Q: 这句代码 `result = sentence[:]` 实现了什么

> 这是python中用来复制列表的一种方式。你使用了列表的分割切片语法 `[:]`，得到列表从第一个到最后一个元素的切片。

Q: 这个脚本很难跑起来啊

> 你需要输入这些代码并保证它能运行。这个脚本可能会有一些小问题，但是它并不复杂。试着用你到目前为止学到的东西来调试脚本，每输入一行，确认一下是否与我的代码一样，并在网上搜索你所不了解的所有问题。

Q: 这对我来说太难了！

> 你可以的，慢慢来，如果需要的话，你逐个字符的输入，然后弄明白它是做什么的。

练习42.对象、类、以及从属关系

有一个重要的概念你需要弄明白，那就是“类(class)”和“对象(object)”的区别。问题在于，class 和 object 并没有真正的不同。它们其实是同样的东西，只是在不同的时间名字不同罢了。我用禅语来解释一下吧：

```
鱼和三文鱼有什么区别？
```

这个问题有没有让你有点晕呢？说真的，坐下来想一分钟。我的意思是说，鱼和三文鱼是不一样，不过它们其实也是一样的是不是？三文鱼是鱼的一种，所以说没什么不同，不过三文鱼又有些特别，它和别的种类的鱼的确不一样，比如三文鱼和大比目鱼就不一样。所以三文鱼和鱼既相同又不同。怪了。

这个问题让人晕的原因是大部分人不会这样去思考问题，其实每个人都懂这一点，你无须去思考鱼和三文鱼的区别，因为你知道它们之间的关系。你知道三文鱼是鱼的一种，而且鱼还有别的种类，根本就没必要去思考这类问题。

让我们更进一步，假设你有一只水桶，里边有三条三文鱼。假设你的好人卡多到没地方用，于是你给它们分别取名叫Frank, Joe,和Mary。现在想想这个问题：

```
Mary和三文鱼有什么区别？
```

这个问题一样的奇怪，但比起鱼和三文鱼的问题来还好点。你知道Mary是一条三文鱼，所以他并没什么不同，他只是三文鱼的一个“实例(instance)”。Frank和Joe一样也是三文鱼的实例。我的意思是说，它们是由三文鱼创建出来的，而且代表着和三文鱼一样的属性。

所以我们的思维方式是（你可能会有点不习惯）：鱼是一个“类(class)”，三文鱼是一个“类(class)”，而Mary是一个“对象(object)”。仔细想想，然后我再一点一点慢慢解释给你。

鱼是一个“类”，表示它不是一个真正的东西，而是一个用来描述具有同类属性的实例的概括性词汇。 你有鳍？你有鳔？你住在水里？好吧那你就是一条鱼。

后来河蟹养殖专家路过，看到你的水桶，于是告诉你：“小伙子，你这些鱼是三文鱼。” 并且专家还定义了一个新的叫做“三文鱼”的“类”，而这个“类”又有它特定的属性。长鼻子？浅红色的肉？生活在海洋里？吃起来味道还可以？那你就是一条三文鱼。

最后一个厨师过来了，他跟专家说：“非也非也，你看到的是三文鱼，我看到的是Mary，而且我要把Mary和剁椒配一起做一道小菜。”于是你就有了一只叫做Mary的三文鱼的“实例(instance)”（三文鱼也是鱼的一个“实例”），并且你使用了它，这样它就是一个“对象(object)”。

这会你应该了解了：Mary是三文鱼的成员，而三文鱼又是鱼的成员。这里的关系式：对象属于某个类，而某个类又属于另一个类。

写成代码是什么样子

这个概念有点绕，不过实话说，你只要在创建和使用 class 的时候操心一下就可以了。我来给你两个区分 Class 和 Object 的小技巧。

首先针对类和对象，你需要学会两个说法，"is-a(是啥)"和"has-a(有啥)"。"是啥"要用在谈论"两者以类的关系互相关联"的时候，而"有啥"要用在"两者无共同点，仅是互为参照"的时候。

接下来，通读这段代码，将每一个注释为 `##` ?? 的位置标明他是"is-a"还是"has-a"的关系，并讲明白这个关系是什么。在代码的开始我还举了几个例子，所以你只要写剩下的就可以了。

记住，"是啥"指的是鱼和三文鱼的关系，而"有啥"指的是三文鱼和鳃的关系。

```
## Animal is-a object (yes, sort of confusing) look at the extra credit
class Animal(object):
    pass

## ??
class Dog(Animal):

    def __init__(self, name):
        ## ??
        self.name = name

## ??
class Cat(Animal):

    def __init__(self, name):
        ## ??
        self.name = name

## ??
class Person(object):

    def __init__(self, name):
        ## ??
        self.name = name

        ## Person has-a pet of some kind
        self.pet = None

## ??
class Employee(Person):

    def __init__(self, name, salary):
        ## ?? hmm what is this strange magic?
        super(Employee, self).__init__(name)
        ## ??
        self.salary = salary

## ??
class Fish(object):
    pass

## ??
class Salmon(Fish):
    pass
```

```
## ??
class Halibut(Fish):
    pass

## rover is-a Dog
rover = Dog("Rover")

## ??
satan = Cat("Satan")

## ??
mary = Person("Mary")

## ??
mary.pet = satan

## ??
frank = Employee("Frank", 120000)

## ??
frank.pet = rover

## ??
flipper = Fish()

## ??
crouse = Salmon()

## ??
harry = Halibut()
```

关于 class Name(object)

记得我曾经强迫让你使用 `class Name(object)` 却没告诉你为什么吧，现在你已经知道了“类”和“对象”的区别，我就可以告诉你原因了。如果我早告诉你的话，你可能会晕掉，也学不会这门技术了。

真正的原因是在 Python 早期，它对于 `class` 的定义在很多方面都是严重有问题的。当他们承认这一点的时候已经太迟了，所以逼不得已，他们需要支持这种有问题的 `class` 。为了解决已有的问题，他们需要引入一种“新类”，这样的话“旧类”还能继续使用，而你也有一个新的正确的类可以使用了。

这就用到了“类即是对象”的概念。他们决定用小写的“object”这个词作为一个类，让你在创建新类时从它继承下来。有点晕了吧？一个类从另一个类继承，而后者虽然是个类，但名字却叫“object”……不过在定义类的时候，别忘记要从 object 继承就好了。

的确如此。一个词的不同就让这个概念变得更难理解，让我不得不现在才讲给你。现在你可以试着去理解“一个是对象的类”这个概念了，如果你感兴趣的话。

不过我还是建议你别去理解了，干脆完全忘记旧格式和新格式类的区别吧，就假设 Python 的 class 永远都要求你加上 (object) 好了，你的脑力要留着思考更重要的问题。

附加题

> 1. 研究一下为什么Python添加了这个奇怪的叫做`object`的类，它究竟有什么含义呢？
> 2. 有没有办法把`Class`当作`Object`使用呢？
> 3. 在习题中为 animals、fish、还有 people 添加一些函数，让它们做一些事情。看看当函数在 Animal 这样的“基类(base class)”里和在 Dog 里有什么区别。
> 4. 找些别人的代码，理清里边的“是啥”和“有啥”的关系。
> 5. 使用列表和字典创建一些新的一对应多的“has-many”的关系。
> 6. 你认为会有一种“has-many”的关系吗？阅读一下关于“多重继承(multiple inheritance)”的资料，然后尽量避免这种用法。

常见问题

Q: 这些注释符`## ??`是干什么的？

> 这些是你需要完成的“填空”，你需要填上 "is-a"或者 "has-a"。再读一遍本节练习，看看其他的注释，弄明白我再说什么。

Q: `self.pet = None`是什么意思？

> 确保给`self.pet`设置了一个默认值`None`

Q: `super(Employee, self).__init__(name)`实现了什么？

> 这是用来执行父类的`__init__`方法的，上网搜索一下“python super”相关文档，阅读文档的各种建议。

练习43.基本的面向对象的分析和设计

在这节练习，我想给你介绍一个使用python创建某类东西的过程，也就是“面向对象编程”（OOP）。我把它叫做一个过程，是因为我将给出一系列按顺序进行的步骤，但是你也不应该死板的遵循这个步骤，企图用它解决所有难题。它们对于许多编程问题只是一个良好的开头，而不应该被认为是解决这些问题的唯一方法。这个过程只是一个你可以遵循的方法：

> 1. 写出或画出你的问题
> 2. 从1中提炼关键问题并搜索相关资料
> 3. 为2中的问题创建一个有层次结构的类和对象映射
> 4. 编写类和测试代码，并保证他们运行
> 5. 重复并精炼

按照这个顺序执行流程，叫做“自顶向下”的方式，意思是说，它从非常抽象宽松的想法开始，然后慢慢提炼，直到想法是坚实的东西，然后你再开始编码。

首先我只是写出这个问题，并试图想出任何我想要的东西就可以了。也许我会画一两个图表，或者某种可能的地图，甚至给自己写了一系列的电子邮件描述这个问题。这给了我表达了问题的关键概念的方法，并探测出我对于这个游戏有什么具体的想法和了解。

接下来，我浏览这些笔记，图表以及描述，通过这些记录我找到我需要的关键问题点。有一个简单的技巧：简单地列出你的笔记和图表中所有的名词和动词,然后写出他们是如何相关的。这一步其实也我为我下一步要写的类、对象、以及函数等提供了命名列表。我利用这个概念列表，研究任何我不明白的地方，如果我需要，我还可以进一步完善它们。

一旦我完成这个列表，我可以创建的一个简单的轮廓/树用来说明这些概念之间的关系。你还可以对着你的名词列表并询问“这个名词和其他的是一个概念吗？或者它们有一个通用的父类吗，那它们的父类是什么？”持续检查，直到你得到一个有层次结构的类，它应该像一棵简单的树或者图表。然后检查你的动词列表，看它们是否可以作为函数的名字添加到你的类树种。

随着这棵类树的生成，我坐下来写一些基本的框架代码，代码中只包括刚才提到的类，类中包含的函数。然后我再写一个测试用例，用来检验我刚才写的类是正确的。有时候我可能只需要在开始写一段测试代码，但是更多的时候，我需要写一段测试，再写一段代码，再写一段测试...直到整个项目完成。

最后，在我完成更多的工作之前，我周期性的重复和精炼这个流程，是他变得清楚和易于理解。如果我在一个没有预料到的地方被一些概念或问题卡住，我会停下来在这一小部分上投入更大的精力来分析解决，直到我解决了这问题，再继续下去。

这节练习中，我将通过建造一个游戏引擎带大家学习这一流程。

分析一个简单的游戏引擎

我打算制作一个叫做 "Gothons from Planet Percal #25"的游戏，这个一个小型的太空冒险游戏。

写或画出这个问题

我为这个游戏写了一小段描述：

“外星人乘坐一个宇宙飞船入侵,我们的英雄需要通过一个迷宫似的房间击败他们,然后他才能逃入一个逃生舱到达下面的行星。游戏将更像一个有着文本输出和有趣的死法的Zork或冒险类型游戏。游戏将包括一个引擎运行充满房间或场景的地图。 当玩家进入房间，每个房间将打印自己的描述,然后告诉引擎运行下一个地图。”

我先描述每个场景：

Death: 这是玩家死亡的场景，应该是有趣的。

Central Corridor: 这是游戏的起点，在这里已经有一个外星人等待英雄的到来，它们需要被一个玩笑打败。

Laser Weapon Armory: 这是英雄得到了中子弹获得的逃生舱之前要炸毁的船。它有一个需要英雄猜数的键盘。

The Bridge: 和外星人战斗的另一个场景，英雄防止炸弹的地方。

Escape Pod: 英雄逃脱的场景，但是需要英雄找对正确的逃生舱

现在，我可以绘制出它们的地图，或者对每个房间再多写一些描述，或者其他一些我脑中出现的想法。

提取关键概念并研究他们

现在我已经有足够多的信息还提取出名词，并分析它们的类结构。首先我列出所有的名词：

> - Alien
> - Player
> - Ship
> - Maze
> - Room
> - Scene
> - Gothon
> - Escape Pod
> - Planet
> - Map
> - Engine
> - Death
> - Central Corridor
> - Laser Weapon Armory
> - The Bridge

我也可能要列出所有的动词，看它们能否作为函数的名字，不过现在我要先跳过这一步。

你需要研究所有你现在还不知道的概念。例如，我可能会玩几个这种类型的游戏，并确保知道他们是如何工作的。我可能会研究船舶和炸弹的设计和工作原理。也许我会研究怎么样将游戏中的数据或状态存储在数据库等一些技术上的问题。我做完这个研究之后，我可能会回到第1步重新开始，根据新的信息，重写描述和提取新的概念。

创建一个层次结构的类和对象映射的概念

当我完成上面的步骤，我通过思考“哪些是类似于其他东西的”，把名词列表转换成一个有层次结构的类树，同样，我也会思考，哪些单词是其他东西的基础呢？

马上我发现，"Room" 和 "Scene"对于我要做的事情来说基本上是一样的事情。在这个游戏中，我选择使用"Scene" 。接下来我发现，所有的特殊房间比如"Central Corridor"基本上也跟"Scene"是一样的。我发现"Death"也是一个"Scene", 由于我选择了"Scene"而不是"Room"，你可以有一个“死亡场景”，而是不一个很奇怪的“死亡房间”。"Maze"和"Map"基本一样，所以我选择使用"Map"，因为我更多的时候都用它。我不想做一个战斗系统，所以我会先忽略"Alien"和 "Player"，但是会把他们保存下来以供以后使用。"Planet"也可能仅仅是另一个场景，而不是什么具体的事情。

在此之后，我开始创建一个层次结构的类：

> - Map
> - Engine
> - Scene
>
> - Death
> - Central Corridor
> - Laser Weapon Armory
> - The Bridge
> - Escape Pod

然后，我会找出说明中每个动词都需要什么行动。比如，我从说明中得知，我需要一个方法来"run"这个引擎，通过地图获得下一个场景"get the next scene"，获得"opening scene"，或者"enter"一个场景。我把这些加到类树里：

> - Map- next_scene- opening_scene
> - Engine- play
> - Scene- enter *Death* Central Corridor *Laser Weapon Armory* The Bridge* Escape Pod

注意一下，我只是把`-enter`放到`Scene`下面，因为我知道所有的场景都会继承它并重写它。

编码类和测试代码并运行它们

当我有了这个类树和一些功能之后，我在编辑器中打开源文件，并尝试编写代码。通常，我会复制粘贴这棵类树到源文件中，然后编辑它。下面是一个如何编码的小例子，文件末尾包含一个简单的测试用例：

```
class Scene(object):

    def enter(self):
        pass

class Engine(object):

    def __init__(self, scene_map):
        pass

    def play(self):
        pass

class Death(Scene):

    def enter(self):
        pass

class CentralCorridor(Scene):

    def enter(self):
        pass

class LaserWeaponArmory(Scene):

    def enter(self):
        pass

class TheBridge(Scene):

    def enter(self):
        pass

class EscapePod(Scene):

    def enter(self):
        pass

class Map(object):

    def __init__(self, start_scene):
        pass

    def next_scene(self, scene_name):
        pass

    def opening_scene(self):
        pass

a_map = Map('central_corridor')
a_game = Engine(a_map)
a_game.play()
```

在这个文件中，你可以看到我只是复制了我想要的层次结构,然后一点点的补齐代码再运行它,看看它在这个基本结构中是否运行顺利。 在这节练习后面的部分，你会填补这段代码的其余部分，使其正常工作，以配合练习开头的游戏描述。

重复并精炼

在我提供的流程中，最后一步并不是实际意义上的一步，而是要做一个循环。在编程的世界里，你永远做不到一次通过，相反，你退回整个过程，并再次根据你从后面的步骤中了解到的信息完善它。有时候我已经到了第3步，但是我发现我还需要在第1、2步做更多工作，我会

停下来并返回去完善它。有时候我也会突然灵光一闪，跳到最后，用我脑子里更好的解决方案编码实现，但是之后，我仍然会回去完成前面的步骤，以确保我的工作覆盖了所有的可能性。

在这一过程的另一个观点是，你不是仅在一个层面上使用这个流程，当你遇到某些特定问题的时候，你可以在任意一个层级上使用该流程。比方说，我不知道如何写 `Engine.play` 方法，我可以静下心来用这个流程只做这一种功能，直到弄清楚这个方法怎么写。

自顶向下和自下而上

因为这个流程在最抽象的概念（顶部）开始，然后再下降到实际执行过程中，因此这一流程通常标示为“自上而下”。我希望你使用这一流程，但你也应该知道，还有另一种方式来解决程序中的问题，这种方式是从代码开始，再“上升”到抽象的概念问题。这种方式被称为“自下而上”。下面是自下而下方式所遵循的步骤：

> 1. 取一小块问题，编写一些代码，并让他勉强运行
> 2. 完善代码，将其转换成一些更正式的包含类和自动化测试的代码。
> 3. 提取其中的关键概念，并尝试找出研究他们。
> 4. 写出到底发生了什么的描述。
> 5. 继续完善代码，也可能是把它扔掉，并重新开始。
> 6. 移动到其他问题上，重复步骤。

当你需要更优质的代码，并在代码中更自然的思考你要解决的问题时，这种方式更好一些。尤其是当你知道小块的难题,但没有足够的信息把握整个概念的时候，这个方式是一个解决问题很好的办法。将问题分解成碎片并探索代码,直到你解决这个问题。然而，你解决问题的途径可能是缓慢而曲折的，所以，我的这一流程中也包含后退并反复研究问题，直到你通过自己所为解决所有难题。

“来自 Percal 25 号行星的哥顿人”的代码

我要告诉你我解决上述问题的最终办法,但我不希望你只是直接跳到这里并输入代码。我希望你能通过我的描述，完成我写的那个简单粗糙的代码框架，并让他运行起来。当你有了自己的解决方案时，你可以回来看看我是怎么解决的。

我准备把文件 `ex43.py` 折成小块，再一一解释，而不是一次性提供完整的代码。

```
from sys import exit
from random import randint
```

这只是我们游戏需要导入的包，没什么新奇的。

```
class Scene(object):

    def enter(self):
        print "This scene is not yet configured. Subclass it and implement enter()."
        exit(1)
```

正如你在框架代码里看到的，我创建了基础类 `Scene` ,它将包含所有scene要做的所有的事。在这段代码中，它们并没有做什么，所以这只是给你的一个如果创建基类的示范。

```
class Engine(object):

    def __init__(self, scene_map):
        self.scene_map = scene_map

    def play(self):
        current_scene = self.scene_map.opening_scene()
        last_scene = self.scene_map.next_scene('finished')

        while current_scene != last_scene:
            next_scene_name = current_scene.enter()
            current_scene = self.scene_map.next_scene(next_scene_name)

        # be sure to print out the last scene
        current_scene.enter()
```

我同样创建了类 `Engine` ，并且你能看到我已经使用了方法 `Map.opening_scene` 和 `Map.next_scene` 。因为在我写 `Map` 类之前，做了一点计划，我可以假设我后面会写这些,然后提前使用它们。

```
class Death(Scene):

    quips = [
        "You died.  You kinda suck at this.",
         "Your mom would be proud...if she were smarter.",
         "Such a luser.",
         "I have a small puppy that's better at this."
    ]

    def enter(self):
        print Death.quips[randint(0, len(self.quips)-1)]
        exit(1)
```

我的第一个场景是一个叫做 `Death` 的场景，它给你展现了你能写的最简单的场景。

```
class CentralCorridor(Scene):

    def enter(self):
        print "The Gothons of Planet Percal #25 have invaded your ship and destroyed"
        print "your entire crew.  You are the last surviving member and your last"
        print "mission is to get the neutron destruct bomb from the Weapons Armory,"
        print "put it in the bridge, and blow the ship up after getting into an "
        print "escape pod."
        print "\n"
        print "You're running down the central corridor to the Weapons Armory when"
        print "a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown co
stume"
        print "flowing around his hate filled body.  He's blocking the door to the"
        print "Armory and about to pull a weapon to blast you."

        action = raw_input("> ")

        if action == "shoot!":
            print "Quick on the draw you yank out your blaster and fire it at the Goth
on."
            print "His clown costume is flowing and moving around his body, which thro
ws"
            print "off your aim.  Your laser hits his costume but misses him entirely.
  This"
            print "completely ruins his brand new costume his mother bought him, which
"
            print "makes him fly into an insane rage and blast you repeatedly in the f
ace until"
            print "you are dead.  Then he eats you."
            return 'death'

        elif action == "dodge!":
            print "Like a world class boxer you dodge, weave, slip and slide right"
            print "as the Gothon's blaster cranks a laser past your head."
            print "In the middle of your artful dodge your foot slips and you"
            print "bang your head on the metal wall and pass out."
            print "You wake up shortly after only to die as the Gothon stomps on"
            print "your head and eats you."
            return 'death'

        elif action == "tell a joke":
            print "Lucky for you they made you learn Gothon insults in the academy."
            print "You tell the one Gothon joke you know:"
            print "Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr, fur fvgf neb
haq gur ubhfr."
            print "The Gothon stops, tries not to laugh, then busts out laughing and c
an't move."
            print "While he's laughing you run up and shoot him square in the head"
            print "putting him down, then jump through the Weapon Armory door."
            return 'laser_weapon_armory'

        else:
            print "DOES NOT COMPUTE!"
            return 'central_corridor'
```

接下来，我创建了`CentralCorridor`，这是游戏的开始。我在写`Map`之前，为游戏制作了这个场景，是因为我后面要引用它们。

```
class LaserWeaponArmory(Scene):

    def enter(self):
        print "You do a dive roll into the Weapon Armory, crouch and scan the room"
        print "for more Gothons that might be hiding.  It's dead quiet, too quiet."
        print "You stand up and run to the far side of the room and find the"
        print "neutron bomb in its container.  There's a keypad lock on the box"
        print "and you need the code to get the bomb out.  If you get the code"
        print "wrong 10 times then the lock closes forever and you can't"
```

```
        print "get the bomb.  The code is 3 digits."
        code = "%d%d%d" % (randint(1,9), randint(1,9), randint(1,9))
        guess = raw_input("[keypad]> ")
        guesses = 0

        while guess != code and guesses < 10:
            print "BZZZZEDDD!"
            guesses += 1
            guess = raw_input("[keypad]> ")

        if guess == code:
            print "The container clicks open and the seal breaks, letting gas out."
            print "You grab the neutron bomb and run as fast as you can to the"
            print "bridge where you must place it in the right spot."
            return 'the_bridge'
        else:
            print "The lock buzzes one last time and then you hear a sickening"
            print "melting sound as the mechanism is fused together."
            print "You decide to sit there, and finally the Gothons blow up the"
            print "ship from their ship and you die."
            return 'death'

class TheBridge(Scene):

    def enter(self):
        print "You burst onto the Bridge with the netron destruct bomb"
        print "under your arm and surprise 5 Gothons who are trying to"
        print "take control of the ship.  Each of them has an even uglier"
        print "clown costume than the last.  They haven't pulled their"
        print "weapons out yet, as they see the active bomb under your"
        print "arm and don't want to set it off."

        action = raw_input("> ")

        if action == "throw the bomb":
            print "In a panic you throw the bomb at the group of Gothons"
            print "and make a leap for the door.  Right as you drop it a"
            print "Gothon shoots you right in the back killing you."
            print "As you die you see another Gothon frantically try to disarm"
            print "the bomb. You die knowing they will probably blow up when"
            print "it goes off."
            return 'death'

        elif action == "slowly place the bomb":
            print "You point your blaster at the bomb under your arm"
            print "and the Gothons put their hands up and start to sweat."
            print "You inch backward to the door, open it, and then carefully"
            print "place the bomb on the floor, pointing your blaster at it."
            print "You then jump back through the door, punch the close button"
            print "and blast the lock so the Gothons can't get out."
            print "Now that the bomb is placed you run to the escape pod to"
            print "get off this tin can."
            return 'escape_pod'
        else:
            print "DOES NOT COMPUTE!"
            return "the_bridge"

class EscapePod(Scene):

    def enter(self):
        print "You rush through the ship desperately trying to make it to"
        print "the escape pod before the whole ship explodes.  It seems like"
        print "hardly any Gothons are on the ship, so your run is clear of"
        print "interference.  You get to the chamber with the escape pods, and"
        print "now need to pick one to take.  Some of them could be damaged"
        print "but you don't have time to look.  There's 5 pods, which one"
        print "do you take?"

        good_pod = randint(1,5)
        guess = raw_input("[pod #]> ")

        if int(guess) != good_pod:
```

```
            print "You jump into pod %s and hit the eject button." % guess
            print "The pod escapes out into the void of space, then"
            print "implodes as the hull ruptures, crushing your body"
            print "into jam jelly."
            return 'death'
        else:
            print "You jump into pod %s and hit the eject button." % guess
            print "The pod easily slides out into space heading to"
            print "the planet below.  As it flies to the planet, you look"
            print "back and see your ship implode then explode like a"
            print "bright star, taking out the Gothon ship at the same"
            print "time.  You won!"

            return 'finished'

class Finished(Scene):

    def enter(self):
        print "You won! Good job."
        return 'finished'
```

这是游戏剩下的场景。因为我知道我需要他们，并且已经想好他们应该如何交织在一起，所以我可以直接编码。

顺便说一下，我不是直接输入这些代码。记得我说过的吗，用增量的方法尝试构建所有的代码，一次只写一小部分。我只是给你看了最终的结果。

```
class Map(object):

    scenes = {
        'central_corridor': CentralCorridor(),
        'laser_weapon_armory': LaserWeaponArmory(),
        'the_bridge': TheBridge(),
        'escape_pod': EscapePod(),
        'death': Death(),
        'finished': Finished(),
    }

    def __init__(self, start_scene):
        self.start_scene = start_scene

    def next_scene(self, scene_name):
        val = Map.scenes.get(scene_name)
        return val

    def opening_scene(self):
        return self.next_scene(self.start_scene)
```

在这之后，我写了 `Map` 类，你可以看到它通过名字把所有的场景存在了字典中。我把这个字典叫做 `Map.scenes` 。这也是为什么map是在scens之后才写的原因，因为这个字典需要包含所有已存在的场景。

```
a_map = Map('central_corridor')
a_game = Engine(a_map)
a_game.play()
```

最后，我让我的代码通过创建一个 `Map` ，并在调用 `play` 之前把map交给 `Engine` ,把游戏运行起来，

你看到的结果

确保你明白这个游戏，并且首先尝试自己解决它。如果你被难住了，可以阅读一小部分我的代码，然后再尝试自己搞定它。

当我运行我的游戏的时候，我可以看到：

```
$ python ex43.py
The Gothons of Planet Percal #25 have invaded your ship and destroyed
your entire crew.  You are the last surviving member and your last
mission is to get the neutron destruct bomb from the Weapons Armory,
put it in the bridge, and blow the ship up after getting into an
escape pod.

You're running down the central corridor to the Weapons Armory when
a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume
flowing around his hate filled body.  He's blocking the door to the
Armory and about to pull a weapon to blast you.
>  dodge!
Like a world class boxer you dodge, weave, slip and slide right
as the Gothon's blaster cranks a laser past your head.
In the middle of your artful dodge your foot slips and you
bang your head on the metal wall and pass out.
You wake up shortly after only to die as the Gothon stomps on
your head and eats you.
Your mom would be proud...if she were smarter.
```

附加题

> 1. 修改这个游戏！你可能不喜欢这个游戏。让游戏运行起来，然后按照你的喜好修改它。这是你的电脑，你可以用它做你想做的事情
> 2. 代码中有一个bug，为什么门锁了11次？
> 3. 解释一下返回至下一个房间的工作原理。
> 4. 增加作弊代码，这样你能通过一些更难的房间。我能只在一行上加两个单词做到这些。
> 5. 回到我的描述和分析，尝试为英雄和他遇见的各种哥顿人创建一个小型作战系统。
> 6. 这其实是一个小版本的“有限状态机(finite state machine)”，找资料阅读了解一下，虽然你可能看不懂，但还是找来看看吧。

常见问题

Q: 我在哪里可以为我的游戏找到故事情节

> 你可以就像给朋友讲述一个故事一样，来创建游戏。或者你可以采取简单的你喜欢的书或电影场景。

练习44.继承Vs.包含

在有关英雄战胜邪恶的童话中，总是有某种形式的黑暗森林。它可能是一个山洞，森林，另一个星球或者其他地方，每个人都知道英雄不应该去。当然，当不就之后坏人被你找到的时候，你发现，英雄已经去那个愚蠢的森林里杀坏人去了。看起来英雄进入了一种状态，这种状态要求英雄必须在这个邪恶的森林中冒险。

在面向对象编程中，继承就是那个黑暗森林。有经验的程序员知道要避免这种邪恶,因为他们知道,黑暗森林深处有着多重继承这个邪恶的皇后。 她喜欢那她那庞大的牙齿吃掉软件和程序员。但这个森林是如此强大，如此诱人，几乎每一个程序员都必须进入它，与邪恶的皇后战斗，尽量做到全身而退，才可以称自己是真正的程序员。你不能抗拒的继承森林的诱惑，所以你进去了。冒险之后，你试着远离那个邪恶的森林，当你再次被迫进入的时候，你会携带一支军队进入（这段翻译的太烂了，实在没办法把编程和童话联系起来！）

这是一种有趣的方式来解说我要在这节练习教你的东西，它叫做继承，当你使用它的时候，一定要当心再当心。正在森林里和女王战斗的程序员可能告诉你你必须进去。他们这么说是因为他们需要你的帮忙，可能因为他们创建了太多他们已经无法掌控的东西。但是你一定要记得：

大多数继承的用途，可以简化或用组合物所取代，并应不惜一切代价避免多重继承。

什么是继承

继承是用来描述一个类从它的父类那里获得大部分甚至全部父类的功能。当你写下 `class Foo(Bar)` 的时候，就发生了继承，这句代码的意思是“创建一个叫做Foo的类，并继承Bar” 。当你执行这句的时候，编程语言使得 `Foo` 的实例所有的行为都跟 `Bar` 的实例一样。这么做，可以让你在类 `Bar` 中放一些通用功能，而那些需要特殊定制的函数或功能可以放在类 `Foo` 中。

当你需要做这些特殊化函数编写的时候，父子类之间有3中交互方法：

> 1. 子类的方法隐性继承父类方法
> 2. 子类重写父类的方法
> 3. 对子类的操作改变父类

我将按顺序给你展示这3种交互方式，并给你看它们的代码。

隐性继承

首先，我将给你展示的是隐性继承发生在你在父类中定义了方法，而在子类中没有：

```
class Parent(object):

    def implicit(self):
        print "PARENT implicit()"

class Child(Parent):
    pass

dad = Parent()
son = Child()

dad.implicit()
son.implicit()
```

在 `class Child` 下使用 `pass` 的目的是告诉Python，在这里你想要一个空白的块。这里创建了一个叫做 `Child` 的类，但是没有在这个类中定义任何性的方法。它将从类 `Parent` 继承获得所有的方法。当你执行这段代码的时候，你会看到：

```
$ python ex44a.py
PARENT implicit()
PARENT implicit()
```

注意一下，尽管我再代码的13行调用了 `son.implicit()` ，而类 `Child` 并没有一个定义叫做 `implicit` 的方法，代码仍然正常工作了，因为它调用了 `Parent` 中定义的同名方法。这个高速你的是：如果你在基类(i.e., `Parent`)中定义了一个方法，那么所有的子类(i.e., `Child`)都可以自动的获得这个功能。对于你需要在很多类中有重复的方法来说，非常方便。

重写方法

关于有些方法是隐式调用的问题原因在于有时候你想让子类有不同的表现。这时你想重写子类中的方法且有效的覆盖父类中的方法。要做到这一点，你只需要在子类中定义一个同名的方法就行，例如

```
class Parent(object):

    def override(self):
        print "PARENT override()"

class Child(Parent):

    def override(self):
        print "CHILD override()"

dad = Parent()
son = Child()

dad.override()
son.override()
```

在这个例子中，两个类中我都有一个叫做 `override` 的方法，所以让我们来看看当你运行此例时都发生了什么

```
$ python ex44b.py
PARENT override()
CHILD override()
```

你可以看到，当第14行执行时，它执行了父类的 `override` 方法，因为变量 `dad` 是一个父类实例，但是当第15行执行时，打印的是子类的 `override` 方法，这是因为 `son` 是一个子类的实例，这个子类重写了那个方法，定义了自己的版本。

休息以下，在继续下面的内容之前，尝试练习这两种方法。

之前或之后改变父类

第三种使用继承的方式比较特别，你想在父类版本的方法执行行为前后给出些提示，你首先像上个例子那样重写了方法，接着你使用一个Python内建的叫做 `super` 的方法得到了父类版本的方法调用。以下这个例子就是这么做的，你可以感受一下上面的描述是什么意思。

```
class Parent(object):

    def altered(self):
        print "PARENT altered()"

class Child(Parent):

    def altered(self):
        print "CHILD, BEFORE PARENT altered()"
        super(Child, self).altered()
        print "CHILD, AFTER PARENT altered()"

dad = Parent()
son = Child()

dad.altered()
son.altered()
```

重要的地方在于9-11行，在 `son.altered()` 被调用前我做了如下操作：

> 1. 因为我已经重写了子类的 `Child.altered` 方法，第9行的运行结果应该和你的期待是一样
> 2. 在这个例子中，我打算使用 `super` 来得到父类的 `Parent.altered` 版本。
> 3. 在第10行，我调用了 `super(Child, self).altered()` 方法, 这个方法能够意识到继承的发生，并给你获得类 `Parent` 。你可以这样读这行代码“调用 `super` ，参数为 `Child` 和 `self` ,然后不管它返回什么，调用方法 `altered` ”
> 4. 在这种情况下，父类的 `Parent.altered` 版本执行，并打印出父类的信息。
> 5. 最后，代码从 `Parent.altered` 返回， `Child.altered` 方法继续打印出剩下的信息。

运行了程序之后，你应该能看到下面的内容：

```
$ python ex44c.py
PARENT altered()
CHILD, BEFORE PARENT altered()
PARENT altered()
CHILD, AFTER PARENT altered()
```

三种组合使用

为了论证上面的三种方式，我有一个最终版本的文件，该文件给你展示了每一种继承方式的交互：

```
class Parent(object):

    def override(self):
        print "PARENT override()"

    def implicit(self):
        print "PARENT implicit()"

    def altered(self):
        print "PARENT altered()"

class Child(Parent):

    def override(self):
        print "CHILD override()"

    def altered(self):
        print "CHILD, BEFORE PARENT altered()"
        super(Child, self).altered()
        print "CHILD, AFTER PARENT altered()"

dad = Parent()
son = Child()

dad.implicit()
son.implicit()

dad.override()
son.override()

dad.altered()
son.altered()
```

通读每一行代码，不管有没有被重写，写一个注释用来解释每一行实现了什么，然后将代码运行起来，确认你的结果和你的期望是否一致：

```
$ python ex44d.py
PARENT implicit()
PARENT implicit()
PARENT override()
CHILD override()
PARENT altered()
CHILD, BEFORE PARENT altered()
PARENT altered()
CHILD, AFTER PARENT altered()
```

使用 `super()` 的原因

这看起来应该是常识，但是随后我们即将进入一个叫做所多重继承的麻烦事。 多重继承是指当你定义一个的类的时候，从一个或多个类继承，例如:

```
class SuperFun(Child, BadStuff):
    pass
```

上述代码的意思是, "创建一个叫做 `SuperFun` 的类，它同时继承类 `Child` 和类 `BadStuff` ."

在这个例子中，只要你隐式的调用任何 `SuperFun` 的实例，Python把必须从类 `Child` 和 `BadStuff` 查询可能的函数，但是，查找也需要一个顺序。为了做到这点，Python使用"方法解析顺序"(MRO)和一种叫做C3的运算法则来直接获得。

因为MRO是复杂的，并使用了明确定义的算法，Python不能让你来获得正确的MRO，相反的，Python提供给你 `super()` 方法，它用来处理所有这一切你需要改变类型的行为，如同我在 `Child.altered` 所实现的。使用 `super()` 你不必担心得到的是否是正确的方法，Python会帮你找到正确的那个。

`__init__` 中使用 `super()`

`super()` 最常见的用途是在基类的 `__init__` 方法里。这通常是你需要在子类里实现什么事情，然后完成父类初始化的地方。以下是在类 `Child` 中这样做的一个简单的例子:

```
class Child(Parent):

    def __init__(self, stuff):
        self.stuff = stuff
        super(Child, self).__init__()
```

除了我在 `__init__` 中初始化父类之前定义了一些变量，这个跟上面的例子 `Child.altered` 几乎是一样的。

包含

继承是有用的， 但另一种方式仅仅是用其他类和模块就做到了同样的事情， 而没有使用隐性继承。如果你看一下使用继承的三种方式，其中的两种方法涉及编写新的代码来替换或改变父类功能。这可以很容易地通过调用模块函数复制。下面是一个例子：

```
class Other(object):

    def override(self):
        print "OTHER override()"

    def implicit(self):
        print "OTHER implicit()"

    def altered(self):
        print "OTHER altered()"

class Child(object):

    def __init__(self):
        self.other = Other()

    def implicit(self):
        self.other.implicit()

    def override(self):
        print "CHILD override()"

    def altered(self):
        print "CHILD, BEFORE OTHER altered()"
        self.other.altered()
        print "CHILD, AFTER OTHER altered()"

son = Child()

son.implicit()
son.override()
son.altered()
```

在这段代码中，我没有使用名字 `Parent` ，因为没有父子的 `is-a` 关系了。这是一个 `has-a` 关系,在这个关系中 `Child``has-a``Other` 被用来保证代码的正常工作。当我运行代码时，看到以下输出：

```
$ python ex44e.py
OTHER implicit()
CHILD override()
CHILD, BEFORE OTHER altered()
OTHER altered()
CHILD, AFTER OTHER altered()
```

你可以看到 `Child` 和 `Other` 中的大部分代码实现了相同的功能。唯一的不同之处在于我必须定义一个 `Child.implicit` 方法去做一个动作. 然后我可以问问自己，如果我需要一个 `Other` 类，是不是只要把它放在一个叫做 `other.py` 的模块中就可以?

什么时候用继承，什么时候用包含

继承与包含的问题可以归结为试图解决可重复使用代码的问题。你不想在你的软件中有重复的代码，因为这不是高效的干净的代码。继承通过创建一种机制，让你在基类中有隐含的功能来解决这个问题。而包含则是通过给你的模块和函数可以在其他类别被调用来解决这个问题。

如果这两种方案都能解决代码复用问题的话，哪一个更合适呢？答案是主观的，这看起来是令人难以相信的，但是我会给你3个指导性原则：

> 1. 不惜一切代价避免多重继承，因为它太复杂太不可靠。如果你必须要使用它，那么一定要知道类的层次结构，并花时间找到每一个类是从哪里来的。
> 2. 将代码封装为模块，这样就可以在许多不同的地方或情况使用。
> 3. 只有当有明显相关的可重用的代码，且在一个共同概念下时，可以使用继承。

不要变成规则的奴隶。关于面向对象编程要记住的是：这是一个程序员创建打包和共享代码的社会习俗。因为它是一个社会习俗，但在Python的法典中，你可能会因为跟你合作的人而被迫避开这些规则。在这种情况下，了解他们是如何使用规则的，然后去适应形势。

附加题

这节练习中只有一个附加题，因为这其实是一个很大的练习。阅读 `http://www.python.org/dev/peps/pep-0008/` 并尝试将它应用到你的代码中。你会发现，有些内容跟你从这本书学到的不同，但是现在你应该能够理解他们的建议，并将其应用到自己的代码中。本书中剩余部分的代码可能会也可能不会遵循这些准则，这个要取决于这些准则是否会使代码更加混乱。我建议你也这样做，因为理解比让大家都对你深奥的知识有印象更重要。

常见问题

Q: 我如何更好的解决我以前没有遇到的问题？

> 更好的解决问题的办法只有一个，那就是尽量多的自己解决遇到的问题。通常，人们遇到一个棘手的问题，就会冲出去寻找答案。当你必须要把事情做好的时候，这种方法很好，但是如果你有时间的话，最好还是自己想办法解决这个问题。尽你所能的思考这个问题，尝试一切可能的办法，直到你解决这个问题。在此之后你找到的答案，你觉得会更加令人满意，而且你还会得到更好的解决问题的方法。

Q: 对象不是复制的类吗？

> 在某些语言里(比如 JavaScript) 是这样的。这些被称为原型的语言，这些语言中对象和类没有什么不同之处。然而在Python中，类作为模板，可以生成新对象，类似于如何使用模具制造硬币。

练习45.你来制作一个游戏

你要开始学会自食其力了。通过阅读这本书你应该已经学到了一点，那就是你需要的所有的信息网上都有，你只要去搜索就能找到。唯一困扰你的就是如何使用正确的词汇进行搜索。学到现在，你在挑选搜索关键字方面应该已经有些感觉了。现在已经是时候了，你需要尝试写一个大的项目，并让它运行起来。

以下是你的需求：

> 1. 制作一个截然不同的游戏。
> 2. 使用多个文件，并使用 `import` 调用这些文件。确认自己知道 `import` 的用法。
> 3. 对于每个房间使用一个 class，class 的命名要能体现出它的用处(例如 `GoldRoom` 、`KoiPondRoom`)。
> 4. 你的执行器代码应该了解这些房间，所以创建一个类来调用并且记录这些房间。有很多种方法可以达到这个目的，不过你可以考虑让每个房间返回下一个房间，或者设置一个变量，让它指定下一个房间是什么。

其他的事情就全靠你了。花一个星期完成这件任务，做一个你能做出来的最好的游戏。使用你学过的任何东西（类，函数，字典，列表......）来改进你的程序。这节课的目的是教你如何构建 class 出来，而这些 class 又能调用到其它 Python 文件中的 class。

我不会详细地告诉你告诉你怎样做，你需要自己完成。试着下手吧，编程就是解决问题的过程，这就意味着你要尝试各种可能性，进行实验，经历失败，然后丢掉你做出来的东西重头开始。当你被某个问题卡住的时候，你可以向别人寻求帮助，并把你的代码贴出来给他们看。如果有人刻薄你，别理他们，你只要集中精力在帮你的人身上就可以了。持续修改和清理你的代码，直到它完整可执行为止，然后再研究一下看它还能不能被改进。

祝你好运，下个星期你做出游戏后我们再见。

评估你的游戏

这节练习的目的是检查评估你的游戏。也许你只完成了一半，卡在那里没有进行下去，也许你勉强做出来了。不管怎样，我们将串一下你应该弄懂的一些东西，并确认你的游戏里有使用到它们。我们将学习如何用正确的格式构建 class，使用 class 的一些通用习惯，另外还有很多的“书本知识”让你学习。

为什么我会让你先行尝试，然后才告诉你正确的做法呢？因为从现在开始你要学会“自给自足”，以前是我牵着你前行，以后就得靠你自己了。后面的习题我只会告诉你你的任务，你需要自己去完成，在你完成后我再告诉你如何可以改进你的作业。

一开始你会觉得很困难并且很不习惯，但只要坚持下去，你就会培养出自己解决问题的能力。你还会找出创新的方法解决问题，这比从课本中拷贝解决方案强多了。

函数的风格

以前我教过的怎样写好函数的方法一样是适用的，不过这里要添加几条：

> - 由于各种各样的原因，程序员将 class (类)里边的函数称作 method （方法）。很大程度上这只是个市场策略（用来推销 OOP），不过如果你把它们称作“函数”的话，是会有啰嗦的人跳出来纠正你的。如果你觉得他们太烦了，你可以告诉他们从数学方面演示一下“函数”和“方法”究竟有什么不同，这样他们会很快闭嘴的。
> - 在你使用 class 的过程中，很大一部分时间是告诉你的 class 如何“做事情”。给这些函数命名的时候，与其命名成一个名词，不如命名为一个动词，作为给 class 的一个命令。就和 list 的 pop (抛出)函数一样，它相当于说：“嘿，列表，把这东西给我 pop 出去。”它的名字不是`remove_from_end_of_list` ，因为即使它的功能的确是这样，这一串字符也不是一个命令。
> - 让你的函数保持简单小巧。由于某些原因，有些人开始学习 class 后就会忘了这一条。

类的风格

> - 你的 class 应该使用 “camel case（驼峰式大小写）”，例如你应该使用 `SuperGoldFactory`而不是`super_gold_factory` 。
> - 你的 `__init__` 不应该做太多的事情，这会让 class 变得难以使用。
> - 你的其它函数应该使用 “underscore format（下划线隔词）”，所以你可以写`my_awesome_hair` ，而不是 `myawesomehair`或者`MyAwesomeHair` 。
> - 用一致的方式组织函数的参数。如果你的 class 需要处理 users、dogs、和 cats，就保持这个次序（特别情况除外）。如果一个函数的参数是`(dog, cat, user)` ，另一个的是`(user, cat, dog)` ，这会让函数使用起来很困难。
> - 不要对全局变量或者来自模组的变量进行重定义或者赋值，让这些东西自顾自就行了。
> - 不要一根筋式地维持风格一致性，这是思维力底下的妖怪喽啰做的事情。一致性是好事情，不过愚蠢地跟着别人遵从一些白痴口号是错误的行为——这本身就是一种坏的风格。好好为自己着想吧。
> - 永远永远都使用`class Name(object)`的方式定义 class，否则你会碰到大麻烦。

代码风格

> - 为了以方便他人阅读，为自己的代码字符之间留下一些空白。你将会看到一些很差的程序员，他们写的代码还算通顺，但字符之间没有任何空间。这种风格在任何编程语言中都是坏习惯，人的眼睛和大脑会通过空白和垂直对齐的位置来扫描和区隔视觉元素，如果你的代码里没有任何空白，这相当于为你的代码上了迷彩装。
> - 如果一段代码你无法朗读出来，那么这段代码的可读性可能就有问题。如你找不到让某个东西易用的方法，试着也朗读出来。这样不仅会逼迫你慢速而且真正仔细阅读过去，还会帮你找到难读的段落，从而知道那些代码的易读性需要作出改进。
> - 学着模仿别人的风格写 Python 程序，直到哪天你找到你自己的风格为止。
> - 一旦你有了自己的风格，也别把它太当回事。程序员工作的一部分就是和别人的代码打交道，有的人审美就是很差。相信我，你的审美某一方面一定也很差，只是你从未意识到而已。
> - 如果你发现有人写的代码风格你很喜欢，那就模仿他们的风格。

好的注释

> - 有程序员会告诉你，说你的代码需要有足够的可读性，这样你就无需写注释了。他们会以自己接近官腔的声音说“所以你永远都不应该写代码注释。”这些人要么是一些顾问型的人物，如果别人无法使用他们的代码，就会付更多钱给他们让他们解决问题。要么他们能力不足，从来没有跟别人合作过。别理会这些人，好好写你的注释。
> - 写注释的时候，描述清楚为什么你要这样做。代码只会告诉你“这样实现”，而不会告诉你“为什么要这样实现”，而后者比前者更重要。
> - 当你为函数写文档注释的时候，记得为别的代码使用者也写些东西。你不需要狂写一大堆，但一两句话写写这个函数的用法还是很有用的。
> - 最后要说的是，虽然注释是好东西，太多的注释就不见得是了。而且注释也是需要维护的，你要尽量让注释短小精悍一语中的，如果你对代码做了更改，记得检查并更新相关的注释，确认它们还是正确的。

为你的游戏评分

现在我要求你假装成我，板起脸来，把你的代码打印出来，然后拿一支红笔，把代码中所有的错误都标出来。你要充分利用你在本章以及前面学到的知识。等你批改完了，我要求你把所有的错误改对。这个过程我需要你多重复几次，争取找到更多的可以改进的地方。使用我前面教过的方法，把代码分解成最细小的单元一一进行分析。

这个练习的目的是训练你对于细节的关注程度。等你检查完自己的代码，再找一段别人的代码用这种方法检查一遍。把代码打印出来，检查出所有代码和风格方面的错误，然后试着在不改坏别人代码的前提下把它们修改正确、

这周我要求你的事情就是批改和纠错，包含你自己的代码和别人的代码，再没有别的了。这节习题难度还是挺大，不过一旦你完成了任务，你学过的东西就会牢牢记在脑中。

练习46.项目骨架

这里你将学会如何建立一个项目“骨架”目录。这个骨架目录具备让项目跑起来的所有基本内容。它里边会包含你的项目文件布局、自动化测试代码，模组，以及安装脚本。当你建立一个新项目的时候，只要把这个目录复制过去，改改目录的名字，再编辑里边的文件就行了。

安装Python 软件包的

你需要使用pip预先安装一些软件包，不过问题就来了。我的本意是让这本书越清晰越干净越好，不过安装软件的方法是在是太多了，如果我要一步一步写下来，那 10 页都写不完，而且告诉你吧，我本来就是个懒人。

所以我不会提供详细的安装步骤了，我只会告诉你需要安装哪些东西，然后让你自己搞定。即使我给了你所需软件详尽的安装说明，你还是不得不与之奋斗。计算机更新换代非常频繁，你在安装过程中遇到问题的时候，可以在网上搜索解决方案。

你需要安装下面的软件包：

> 1. pip – http://pypi.python.org/pypi/pip
> 2. distribute – http://pypi.python.org/pypi/distribute
> 3. nose – http://pypi.python.org/pypi/nose/
> 4. virtualenv – http://pypi.python.org/pypi/virtualenv

不要只是手动下载并且安装这些软件包，你应该看一下别人的建议，尤其看看针对你的操作系统别人是怎样建议你安装和使用的。同样的软件包在不一样的操作系统上面的安装方式是不一样的，不一样版本的 Linux 和 OSX 会有不同，而 Windows 更是不同。

我要预先警告你，这个过程会是相当无趣。在业内我们将这种事情叫做 “yak shaving(剃牦牛)”。它指的是在你做一件有意义的事情之前的一些准备工作，而这些准备工作又是及其无聊冗繁的。你要做一个很酷的 Python 项目，但是创建骨架目录需要你安装一些软件包，而安装软件包之前你还要安装软件包安装工具(package installer)，而要安装这个工具你还得先学会如何在你的操作系统下安装软件，真是烦不胜烦呀。

无论如何，还是克服困难把。你就把它当做进入编程俱乐部的一个考验。每个程序员都会经历这条道路，在每一段“酷”的背后总会有一段“烦”的。

> **NOTE:**有时候python的安装程序不会把 `C:\Python27\Script` 加入到系统的 `PATH` 中，如果你遇到了这个问题，就参照练习0自己把这个目录加上：`[Environment]::SetEnvironmentVariable("Path", "$env:Path;C:\Python27\Scripts", "Use`

创建骨架内容

首先使用下述命令创建你的骨架目录：

```
$ mkdir projects
$ cd projects/
$ mkdir skeleton
$ cd skeleton
$ mkdir bin NAME tests docs
```

我使用了一个叫 `projects` 的目录，用来存放我自己的各个项目。然后我在里边建立了一个叫做 `skeleton` 的文件夹，这就是我们新项目的基础目录。其中叫做 `NAME` 的文件夹是你的项目的主文件夹，你可以将它任意取名。

接下来我们要配置一些初始文件：

```
$ touch NAME/__init__.py
$ touch tests/__init__.py
```

在windows上，你可以这样配置初始文件：

```
$ new-item -type file NAME/__init__.py
$ new-item -type file tests/__init__.py
```

以上命令为你创建了空的模组目录，以供你后面为其添加代码。然后我们需要建立一个 `setup.py` 文件，这个文件在安装项目的时候我们会用到它：

```
try:
    from setuptools import setup
except ImportError:
    from distutils.core import setup

config = {
    'description': 'My Project',
    'author': 'My Name',
    'url': 'URL to get it at.',
    'download_url': 'Where to download it.',
    'author_email': 'My email.',
    'version': '0.1',
    'install_requires': ['nose'],
    'packages': ['NAME'],
    'scripts': [],
    'name': 'projectname'
}

setup(**config)
```

编辑这个文件，把自己的联系方式写进去，然后放到那里就行了。

最后你需要一个简单的测试专用的骨架文件叫 `tests/NAME_tests.py` ：

```
from nose.tools import *
import NAME

def setup():
    print "SETUP!"

def teardown():
    print "TEAR DOWN!"

def test_basic():
    print "I RAN!"
```

最终的目录结构

当你完成一切设置,你的目录应该看起来像我在这里:

```
skeleton/
     NAME/
         __init__.py
     bin/
     docs/
     setup.py
     tests/
         NAME_tests.py
         __init__.py
```

从现在开始，你应该在这个目录下运行命令。如果你不能执行 `ls -R` 命令并看到相似的目录结构，说明你在一个错误的目录下。比如，人们经常会到 `tests/` 目录下尝试执行文件，那肯定是无法运行的。要运行你应用的测试用例，你也应该在目录 `tests/` 的上一层目录执行，加入你这样执行：

```
$ cd tests/   # WRONG! WRONG! WRONG!
$ nosetests

----------------------------------------------------------------------
Ran 0 tests in 0.000s

OK
```

那结果肯定是错误的，你应当在 `tests/` 目录的上一层目录执行，所以为了修正你的错误，你应该这样做：

```
$ cd ..   # get out of tests/
$ ls      # CORRECT! you are now in the right spot
NAME                bin             docs            setup.py        tests
$ nosetests
.
----------------------------------------------------------------------
Ran 1 test in 0.004s

OK
```

一定要记住这一点，因为人们经常犯这个错误。

测试你的配置

安装了所有上面的软件包以后，你就可以做下面的事情了：

```
$ nosetests
.
----------------------------------------------------------------------
Ran 1 test in 0.007s

OK
```

下一节练习中我会告诉你 `nosetests` 的功能，不过如果你没有看到上面的画面，那就说明你哪里出错了。确认一下你的 `NAME` 和 `tests` 目录下存在 `__init__.py`， 并且你没有把 `tests/NAME_tests.py` 命名错。

使用这个项目骨架

剃牦牛的事情已经做的差不多了，以后每次你要新建一个项目时，只要做下面的事情就可以了：

> 1. 拷贝这份骨架目录，把名字改成你新项目的名字。
> 2. 再将 `NAME` 模组更名为你需要的名字，它可以是你项目的名字，当然别的名字也行。
> 3. 编辑 `setup.py` 让它包含你新项目的相关信息。
> 4. 重命名 `tests/NAME_tests.py` ，让它的名字匹配到你模组的名字。
> 5. 使用 `nosetests` 检查有无错误。
> 6. 开始写代码吧。

小测验

本节没有附加题，不过有一些小测验需要你完成

> 1. 找文档阅读，学会使用你前面安装了的软件包。
> 2. 阅读关于 `setup.py` 的文档，看它里边可以做多少配置。Python 的安装器并不是一个好软件，所以使用起来也非常奇怪。
> 3. 创建一个项目，在模组目录里写一些代码，并让这个模组可以运行。
> 4. 在 `bin` 目录下放一个可以运行的脚本，找材料学习一下怎样创建可以在系统下运行的 Python 脚本。
> 5. 在你的 `setup.py` 中加入 `bin` 这个目录，这样你安装时就可以连它安装进去。
> 6. 使用 `setup.py` 安装你的模组，并确定安装的模组可以正常使用，最后使用 `pip` 将其卸载。

常见问题

Q: 这些说明在windows上也是一样的吗？

> 是的，不过也取决于你windows系统的版本，你可能需要在配置上下点功夫它才能正常运行，坚持研究并尝试，直到你能在windows上正常的运行这个骨架，或者你可以找一些有python+windows开发经验的人帮忙。

Q: 我好像不能在windows上运行 `nosetests`

> 有时候python的安装程序不会把 `C:\Python27\Script` 加入到系统的 `PATH` 中，如果你遇到了这个问题，就参照练习0自己把这个目录加上： `[Environment]::SetEnvironmentVariable("Path", "$env:Path;C:\Python27\Scripts", "Use`

Q: 我应该在我的配置文件 `setup.py` 中放些什么？

> 确认你阅读了distutils的文档 `http://docs.python.org/distutils/setupscript.html` 。

Q:我好像不能加载 `NAME` 模块，而且还有个"ImportError"报错

> 确认你创建了 `NAME/__init__.py` 这个文件，如果你用的windows系统，确认你没有把这文件命名为 `NAME/__init__.py.txt` ，很多程序员都犯过这个错。

Q: 我们为什么需要一个 `bin/` 文件夹

> 这只是一个用来存放在命令行执行的脚本的地方，不是用来存在模块的。

Q: 你有一个真实的项目举例吗？

> 有很多python写的项目都可以作为实例，你可以看看我创建的这个简单的项目 `https://gitorious.org/python-modargs` .

Q: 我的 `nosetests` 运行时只显示正在运行一个测试，这是正确的吗？

> 是的，我的也是这么显示的。

练习47.自动化测试

你需要一遍一遍地在你的游戏中输入命令，来测试游戏的功能是否正常。这个过程是很枯燥无味的。如果能写一小段代码用来测试你的代码岂不是更好？然后无论你对程序做了什么修改，或者添加了什么新东西，你只要"跑一下你的测试"，而这些测试能确认程序依然能正确运行。这些自动测试不会抓到所有的 bug，但可以让你无需重复输入命令运行你的代码，从而为你节约很多时间。

从这节练习开始，以后的练习将不会有"你应该看到的结果"部分，取而代之的是一个"你应该测试的东西"。从现在开始，你需要为自己写的所有代码写自动化测试，而这将让你成为一个更好的程序员。

我不会解释你为什么需要写自动化测试。我要告诉你的是，你想要成为一个程序员，而程序的作用是让无聊冗繁的工作自动化，测试一个软件毫无疑问是无聊冗繁的，所以你还是写点代码让它为你测试的更好。

这应该是你需要的所有的解释了。因为你写单元测试的原因是让你的大脑更加强健。读了这本书，你写了很多代码让它们实现一些事情。现在你将更进一步，写出懂得你写的其他代码的代码。这个写代码测试你写的其他代码的过程将强迫你清楚的理解你之前写的代码。这会让你更清晰地了解你写的代码实现的功能及其原理，而且让你对细节的注意更上一个台阶。

编写测试用例

我们将拿一段非常简单的代码为例，写一个简单的测试，这个测试将建立在上节我们创建的项目骨架上面。

首先从你的项目骨架创建一个叫做 `ex47` 的项目。下面是你要采取的步骤。我会给出文字说明，而不是直接告诉你该如何写代码，所以你要自己弄明白并写出代码。

> 1. 复制 `skeleton` 到 `ex47`
> 2. 将所有的 `NAME` 重命名为 `ex47`
> 3. 修改所有文件中 `NAME` 为 `ex47`
> 4. 最后删除所有的 `*.pyc` 文件

如果遇到什么困难，回顾一下练习46，如果你不能简单的完成这些，那你需要多练习几次。

> **NOTE:** 记得通过命令 `nosetests` 来检测你的测试代码没有错误信息。你可以通过 `python ex47_tests.py` 来运行，但是它不会那么容易的运行，你必须保证你的每一个测试文件正常运行。

接下来创建一个简单的 `ex47/game.py` 文件，里边放一些用来被测试的代码。我们现在放一个傻乎乎的小 class 进去，用来作为我们的测试对象：

```
class Room(object):

    def __init__(self, name, description):
        self.name = name
        self.description = description
        self.paths = {}

    def go(self, direction):
        return self.paths.get(direction, None)

    def add_paths(self, paths):
        self.paths.update(paths)
```

准备好了这个文件，接下来把测试骨架改成这样子：

```
from nose.tools import *
from ex47.game import Room

def test_room():
    gold = Room("GoldRoom",
                """This room has gold in it you can grab. There's a
                door to the north.""")
    assert_equal(gold.name, "GoldRoom")
    assert_equal(gold.paths, {})

def test_room_paths():
    center = Room("Center", "Test room in the center.")
    north = Room("North", "Test room in the north.")
    south = Room("South", "Test room in the south.")

    center.add_paths({'north': north, 'south': south})
    assert_equal(center.go('north'), north)
    assert_equal(center.go('south'), south)

def test_map():
    start = Room("Start", "You can go west and down a hole.")
    west = Room("Trees", "There are trees here, you can go east.")
    down = Room("Dungeon", "It's dark down here, you can go up.")

    start.add_paths({'west': west, 'down': down})
    west.add_paths({'east': start})
    down.add_paths({'up': start})

    assert_equal(start.go('west'), west)
    assert_equal(start.go('west').go('east'), start)
    assert_equal(start.go('down').go('up'), start)
```

这个文件导入了你在`ex47.game`创建的`Room`这个类，接下来我们要做的就是测试它。于是我们看到一系列的以`test_`开头的测试函数，它们就是所谓的“测试用例(test case)”，每一个测试用例里面都有一小段代码，它们会创建一个或者一些房间，然后去确认房间的功能和你期望的是否一样。它测试了基本的房间功能，然后测试了路径，最后测试了整个地图。

这里最重要的函数是`assert_equal`，它保证了你设置的变量，以及你在`Room`里设置的路径和你的期望相符。如果你得到错误的结果的话，`nosetests`将会打印出一个错误信息，这样你就可以找到出错的地方并且修正过来。

测试指南

在写测试代码时，你可以照着下面这些不是很严格的指南来做：

> 1. 测试脚本要放到 `tests/` 目录下，并且命名为 `BLAH_tests.py` ，否则 `nosetests` 就不会执行你的测试脚本了。这样做还有一个好处就是防止测试代码和别的代码互相混掉。
> 2. 为你的每一个模组写一个测试。
> 3. 测试用例（函数）保持简短，但如果看上去不怎么整洁也没关系，测试用例一般都有点乱。
> 4. 就算测试用例有些乱，也要试着让他们保持整洁，把里边重复的代码删掉。创建一些辅助函数来避免重复的代码。当你下次在改完代码需要改测试的时候，你会感谢我这一条建议的。重复的代码会让修改测试变得很难操作。
> 5. 最后一条是别太把测试当做一回事。有时候，更好的方法是把代码和测试全部删掉，然后重新设计代码。

你看到的结果

```
$ nosetests
...
----------------------------------------------------------------------
Ran 3 tests in 0.008s

OK
```

如果一切工作正常的话，你看到的结果应该就是这样。试着把代码改错几个地方，然后看错误信息会是什么，再把代码改正确。

附加题

> 1. 仔细读读 `nosetest` 相关的文档，再去了解一下其他的替代方案。
> 2. 了解一下 Python 的 “doc tests” ，看看你是不是更喜欢这种测试方式。
> 3. 改进你游戏里的 Room，然后用它重建你的游戏，这次重写，你需要一边写代码，一边把单元测试写出来。

常见问题

Q: 当我运行 `nosetest` 的时候，我遇到一个语法错误

> 如果你遇到这个报错，看看错误信息是怎么说的，并改正该行或上一行的错误。类似 `nosetest` 的工具是在运行你的代码和测试代码，所以他可以像运行python一样发现你的语法错误。

Q: 我无法导入`ex47.game`

> 确认你创建了`ex47/__init__.py`文件，参照练习46，看它是如何做的。如果这个文件没有问题，在OSX/Linux下执行:`export PYTHONPATH=.`在window下执行`$env:PYTHONPATH = "$env:PYTHONPATH;."`，最后，确认你是用`nosetest`运行测试脚本，而不是用python。

Q: 我运行`nosetest`的时候，遇到一个报错`UserWarning`

> 你可能安装了两个版本的python或者没有使用`distribute`，按照我在练习46中所描述的安装`distribute`或`pip`。

练习48.更复杂的用户输入

在以前的游戏中，你只是设置一些简单的预定义字符串作为用户输入处理，用户输入“run”，程序能正常运行，但是你输入“run fast”，程序就会运行失败。我们需要一个设备，它可以识别用户以各种方式输入的语汇。例如下面的机种表述都应该被支持才对：

> - open door
> - open the door
> - go THROUGH the door
> - punch bear
> - Punch The Bear in the FACE

也就是说，如果用户的输入和常用英语很接近也应该是可以的，而你的游戏要识别出它们的意思。为了达到这个目的，我们将写一个模块专门做这件事情。这个模组里边会有若干个类，它们互相配合，接受用户输入，并且将用户输入转换成你的游戏可以识别的命令。

英语的简单格式是这个样子的：

> - 单词由空格隔开。
> - 句子由单词组成。
> - 语法控制句子的含义。

以最好的开始方式是先搞定如何得到用户输入的词汇，并判断出它们是什么。

我们的游戏词典

我在游戏里创建了下面这些语汇：

> - 表示方向: north, south, east, west, down, up, left, right, back.
> - 动词: go, stop, kill, eat.
> - 修饰词: the, in, of, from, at, it
> - 名词: door, bear, princess, cabinet.
> - 数字: 由 0-9 构成的数字。

说到名词，我们会碰到一个小问题，那就是不一样的房间会用到不一样的一组名词，不过让我们先挑一小组出来写程序，以后再做改进。

如何断句

我们已经有了词汇表，为了分析句子的意思，接下来我们需要找到一个断句的方法。我们对于句子的定义是“空格隔开的单词”，所以只要这样就可以了：

```
stuff = raw_input('> ')
words = stuff.split()
```

目前做到这样就可以了，不过这招在相当一段时间内都不会有问题。

词汇元组

一旦我们知道了如何将句子转化成词汇列表，剩下的就是逐一检查这些词汇，看它们是什么类型。为了达到这个目的，我们将用到一个非常好使的 Python 数据结构，叫做”元组(tuple)”。元组其实就是一个不能修改的列表。创建它的方法和创建列表差不多，成员之间需要用逗号隔开，不过方括号要换成圆括号 `()` ：

```
first_word = ('verb', 'go')
second_word = ('direction', 'north')
third_word = ('direction', 'west')
sentence = [first_word, second_word, third_word]
```

这样我们就创建了一个(TYPE,WORD)组，让你识别出单词，并且对它执行指令。

这只是一个例子，不过最后做出来的样子也差不多。你接受用户输入，用 `split` 将其分隔成单词列表，然后分析这些单词，识别它们的类型，最后重新组成一个句子。

扫描输入

现在你要写的是词汇扫描器。这个扫描器会将用户的输入字符串当做参数，然后返回由多个(TOKEN, WORD) 组成的一个列表，这个列表实现类似句子的功能。如果一个单词不在预定的词汇表中，那它返回时 WORD 应该还在，但 TOKEN 应该设置成一个专门的错误标记。这个错误标记将告诉用户哪里出错了。

有趣的地方来了。我不会告诉你这些该怎样做，但我会写一个“单元测试(unit test)”，而你要把扫描器写出来，并保证单元测试能够正常通过。

异常和数字

有一件小事情我会先帮帮你，那就是数字转换。为了做到这一点，我们会作一点弊，使用“异常(exceptions)”来做。“异常”指的是你运行某个函数时得到的错误。你的函数在碰到错误时，就会“抛出(raise)”一个“异常”，然后你就要去处理(handle)这个异常。假如你在Python 里写了这些东西：

```
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> int("hell")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hell'
```

这个`ValueError`就是`int()`函数抛出的一个异常。因为你给`int()`的参数不是一个数字。`int()`函数其实也可以返回一个值来告诉你它碰到了错误，不过由于它只能返回整数值，所以很难做到这一点。它不能返回-1，因为这也是一个数字。`int()`没有纠结在它“究竟应该返回什么”上面，而是提出了一个叫做`ValueError`的异常，然后你只要处理这个异常就可以了。

处理异常的方法是使用`try`和`except`这两个关键字：

```
def convert_number(s):
    try:
        return int(s)
    except ValueError:
        return None
```

你把要试着运行的代码放到`try`的区段里，再将出错后要运行的代码放到`except`区段里。在这里，我们要试着调用`int()`去处理某个可能是数字的东西，如果中间出了错，我们就抓到这个错误，然后返回`None`。

在你写的扫描器里面，你应该使用这个函数来测试某个东西是不是数字。做完这个检查，你就可以声明这个单词是一个错误单词了。

测试第一的挑战

测试首先是一种编程策略，你先写一段自动化测试代码，假装代码是在正常运行的，然后你再写出代码保证测试代码能正常运行。这种方法用在当你不知道代码是如何运行，但又可以想象必须使用它的时候。比如说，如果你知道你需要在另一个模块中使用一个新类,但是你不太知道如何实现这个类，那么先写出测试程序。

我将给你一份测试代码，你需要写出代码，保证测试代码能正常工作。为了完成这个任务，你可以看看下面的流程：

> 1. 创建一小部分我给你的测试代码
> 2. 确保它运行失败,你知道测试实际上是确认功能的工作原理。
> 3. 到你的源代码文件`lexicon.py`中，写出能使测试代码通过的代码
> 4. 重复以上工作直到你实现测试中的所有点

当你做到3的时候，和其他编写代码的方法相结合也是很好的方法：

> 1. 编写你需要的函数或类的基本框架
> 2. 添加注释，解释说明这个函数是如何运行的
> 3. 按照描述中的注释写代码
> 4. 去掉注释

这种写代码的方法被称作“psuedo code”，用在你不知道该如何实现某些功能，但是会用自己的语言来描述这个功能的时候。

结合“test first”和“psuedo code”策略，我们得出一个编程的简易流程：

> 1. 写一些运行失败的测试用例
> 2. 写出测试要用的函数、方法、类的基本结构
> 3. 用自己的语言填充这些框架，解释它们的功能
> 4. 用代码替换注释，直到测试代码运行通过
> 5. 重复

在这节练习中，你将通过运行我给你的测试程序逆向运行 `lexicon.py` 来实践这个方法。

你应该测试的东西

这里是你要用到的测试文件：

```
from nose.tools import *
from ex48 import lexicon

def test_directions():
    assert_equal(lexicon.scan("north"), [('direction', 'north')])
    result = lexicon.scan("north south east")
    assert_equal(result, [('direction', 'north'),
                          ('direction', 'south'),
                          ('direction', 'east')])

def test_verbs():
    assert_equal(lexicon.scan("go"), [('verb', 'go')])
    result = lexicon.scan("go kill eat")
    assert_equal(result, [('verb', 'go'),
                          ('verb', 'kill'),
                          ('verb', 'eat')])

def test_stops():
    assert_equal(lexicon.scan("the"), [('stop', 'the')])
    result = lexicon.scan("the in of")
    assert_equal(result, [('stop', 'the'),
                          ('stop', 'in'),
                          ('stop', 'of')])

def test_nouns():
    assert_equal(lexicon.scan("bear"), [('noun', 'bear')])
    result = lexicon.scan("bear princess")
    assert_equal(result, [('noun', 'bear'),
                          ('noun', 'princess')])

def test_numbers():
    assert_equal(lexicon.scan("1234"), [('number', 1234)])
    result = lexicon.scan("3 91234")
    assert_equal(result, [('number', 3),
                          ('number', 91234)])

def test_errors():
    assert_equal(lexicon.scan("ASDFADFASDF"), [('error', 'ASDFADFASDF')])
    result = lexicon.scan("bear IAS princess")
    assert_equal(result, [('noun', 'bear'),
                          ('error', 'IAS'),
                          ('noun', 'princess')])
```

你需要用项目框架写出一个新的项目，就像你在练习47中做的一样。然后你需要创建这个测试用例以及你会用到的 `lexicon.py` ，看看测试用例顶部，看看它是如何被导入的。

接下来，按照我给你的提示写一些测试用例。看看我是如何做的：

> 1. 在测试用例顶部写上导入（import），并保证它正常运行
> 2. 创建第一个测试用例 `test_directions` 的空版本，并保证它正常运行
> 3. 写出测试用例 `test_directions` 的第一行，保证它运行失败
> 4. 到 `lexicon.py` 文件，创建一个空的 `scan` 方法
> 5. 运行测试用例，至少保证 `scan` 方法运行，即便测试用例运行失败
> 6. 为 `scan` 写出伪代码注释，用来说明 `scan` 如何通过 `test_directions` 测试
> 7. 写出与注释相匹配的代码，保证 `test_directions` 测试通过
> 8. 回到方法 `test_directions` ，写完剩下的行
> 9. 回到 `lexicon.py` 中的 `scan` 方法，补全代码直到 `test_directions` 测试通过
> 10. 这样，当你的第一个测试通过，你移动到下一个测试重复以上步骤。

只要你坚持在每次执行此过程中的一小块，你可以成功将大问题分解成更小的问题来解决。就像爬山的时候，你把整段路程分成一小段一小段。

附加题

> 1. 改进单元测试，让它覆盖到更多的语汇。
> 2. 向语汇列表添加更多的语汇，并且更新单元测试代码。
> 3. 让你的扫描器能够识别任意大小写的词汇。更新你的单元测试。
> 4. 找出另外一种转换为数字的方法。
> 5. 我的解决方案用了 37 行代码，你的是更长还是更短呢？

常见问题

Q: 为什么我一直有这个报错 `ImportErrors` ？

> 导入异常通常有以下几点原因：1，在你的模块（modules）目录下没有生成`__init__.py`文件；2，你在错误的目录下启动服务；3，你导入的模块有拼写错误；4，你的`PYTHONPATH`没有设置成`.`。

Q: `try-except` 和 `if-else` 有什么区别？

> `try-expect`是用来处理模块抛出的异常，永远都不能用`if-else`代替。

Q: 有没有办法能实现在等待用户输入的时候，游戏也一样运行

> 我假设一种情况，你想实现用户在反应不够快的情况下会遭到怪物的攻击，这是可能的，但是它涉及的模块和技术是本书范围之外的。

练习49.写代码语句

从这个小游戏的词汇扫描器中，我们应该可以得到类似下面的列表：

```
python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from ex48 import lexicon
>>> lexicon.scan("go north")
[('verb', 'go'), ('direction', 'north')]
>>> lexicon.scan("kill the princess")
[('verb', 'kill'), ('stop', 'the'), ('noun', 'princess')]
>>> lexicon.scan("eat the bear")
[('verb', 'eat'), ('stop', 'the'), ('noun', 'bear')]
>>> lexicon.scan("open the door and smack the bear in the nose")
[('error', 'open'), ('stop', 'the'), ('error', 'door'), ('error', 'and'), ('error', 's
mack'), ('stop', 'the'), ('noun', 'bear'), ('stop', 'in'), ('stop', 'the'), ('error',
'nose')]
```

现在让我们把它转化成游戏可以使用的东西，也就是一个 `Sentence` 类。

如果你还记得学校学过的东西的话，一个句子是由这样的结构组成的：主语(Subject) + 谓语(动词 Verb) + 宾语(Object)

很显然实际的句子可能会比这复杂，而你可能已经在英语的语法课上面被折腾得够呛了。我们的目的，是将上面的元组列表转换为一个 `Sentence` 对象，而这个对象又包含主谓宾各个成员。

匹配(Match)和窥视(Peek)

为了达到这个效果，你需要四(五)样工具：

> 1. 循环访问元组列表的方法，这挺简单的。
> 2. 匹配我们的主谓宾设置中不同种类元组的方法。
> 3. 一个“窥视”潜在元组的方法，以便做决定时用到。
> 4. 跳过(skip)我们不在乎的内容的方法，例如形容词、冠词等没有用处的词汇。
> 5. 一个用来存放最终结果的 `Sentence` 对象

我们要把这些函数放在一个叫做 `ex48.parser` 的类中，再把这个类放在 `ex48/parser.py` 中，以便于我们能够测试它们。我们使用 `peek` 函数来查看元组列表中的下一个成员，做匹配以后再对它做下一步动作。

句子的语法

在你写代码之前，你要弄明白一个基础的英语句子的语法是如何工作的。在我们的练习中，我们准备创建一个叫做`Sentence`的类，它有如下3个属性：

Sentence.subject（句子的主语）这是任意一个句子的主语，大部分时候可以默认为“玩家player”，比如一个句子“run north 向北跑”, 也就是说 "player run north 玩家向北跑"。主语应该是一个名词。

Sentence.verb（句子的谓语）这就是句子的的作用。 在 "run north" 中，谓语应该是 "run". 谓语应该是一个动词。

Sentence.object（句子的宾语） 这又是一个名词，指的是动词做了什么。在我们游戏中， 我们分辨出的方向就是宾语。在 "run north" 中，单词"north"就是宾语。在 "hit bear" 中，单词"bear" 就是宾语。

我们的程序解析器使用我们给出的函数并返回解析后的句子，转换成一个`list`或`Sentence`对象，用来接收匹配用户输入

关于异常(Exception)

已经简单学过关于异常的一些东西，但还没学过怎样抛出(raise)它们。这节的代码演示了如何raise 前面定义的`ParserError`。注意`ParserError`是一个定义为`Exception`类型的 class。另外要注意我们是怎样使用`raise`这个关键字来抛出异常的。

你的测试代码应该也要测试到这些异常，这个我也会演示给你如何实现。

程序代码

如果你希望更大的挑战，停在这里，然后只听我的描述来完成代码。当你遇到难题的时候，可以再回来看看是我如何做的。不过，尝试自己实现代码功能对你来说真的是个很好的锻炼。我要开始串讲我的代码了，你可以开始在自己的`ex48/parser.py`中输入代码。我们从异常处理开始我们的代码编写：

```
class ParserError(Exception):
    pass
```

这就是你创建一个可以抛出的异常类`ParserError`,接下来，我们需要一个句子类`Sentence`：

```
class Sentence(object):

    def __init__(self, subject, verb, obj):
        # remember we take ('noun','princess') tuples and convert them
        self.subject = subject[1]
        self.verb = verb[1]
        self.object = obj[1]
```

到目前为止，我们没有写什么特别的代码，只是创建了两个简单的类。

在我们的问题描述中，我们需要一个函数用来看到列表中的单词并返回单词的类型：

```
def peek(word_list):
    if word_list:
        word = word_list[0]
        return word[0]
    else:
        return None
```

我们需要这个函数是因为，我们要基于下一个单词来选择确认我们要处理的句子是什么，然后我们可以调用另一个函数来处理这个单词，并将程序继续下去。

我们使用 `match` 函数来处理单词，用它来确认预期中的单词是否是正确的类型，将它移出列表，并返回该词：

```
def match(word_list, expecting):
    if word_list:
        word = word_list.pop(0)

        if word[0] == expecting:
            return word
        else:
            return None
    else:
        return None
```

相当简单是不是，不过还是要确认你理解了这些代码以及为什么我是这么写的。我需要依据我看到的列表中的下一个单词来决定我现在处理的句子的类型，然后再用这个单词创建我的 `Sentence` .

最后，我们需要一个方法来跳过句子中我们不关心的单词。这些单词会被打上“停用词”（stop类型的词）的标签，比如"the","and"以及"a"等：

```
def skip(word_list, word_type):
    while peek(word_list) == word_type:
        match(word_list, word_type)
```

记住 `skip` 不只跳过一个单词而是跳过所有该类型的词，也就是说，如果有人输入了“scream at the bear”，经过处理最后会得到"scream" 和 "bear".

以上是我们分析函数的基本结构，我可以用它们来处理我们需要的任何文本，尽管我们的程序非常简单，剩下的函数也都是非常短的。

首先，我们来完成解析动词的部分：

```
def parse_verb(word_list):
    skip(word_list, 'stop')

    if peek(word_list) == 'verb':
        return match(word_list, 'verb')
    else:
        raise ParserError("Expected a verb next.")
```

我们跳过所有"stop"类型的词，然后提前获得下一个单词，并确认它是"verb"类型，如果不是，则抛出一个异常 `ParserError` 说明为什么不是。如果是"verb"类型，则使用"match"处理，将它移出列表。一个处理"sentence"类的类似函数：

```
def parse_object(word_list):
    skip(word_list, 'stop')
    next_word = peek(word_list)

    if next_word == 'noun':
        return match(word_list, 'noun')
    elif next_word == 'direction':
        return match(word_list, 'direction')
    else:
        raise ParserError("Expected a noun or direction next.")
```

重复操作，跳过"stop"类型的词，提前判断下一个词，决定下一个"sentence".在函数 `parse_object` 中，我们需要同时处理“名词”和类似宾语的“方向”，解析主语的方法也是一样的，但是当我们处理隐藏的名词"player"的时候，我们需要用到"peek"：

```
def parse_subject(word_list):
    skip(word_list, 'stop')
    next_word = peek(word_list)

    if next_word == 'noun':
        return match(word_list, 'noun')
    elif next_word == 'verb':
        return ('noun', 'player')
    else:
        raise ParserError("Expected a verb next.")
```

所有的方式都准备好之后，我们最后一个函数 `parse_sentence` 也是非常简单的：

```
def parse_sentence(word_list):
    subj = parse_subject(word_list)
    verb = parse_verb(word_list)
    obj = parse_object(word_list)

    return Sentence(subj, verb, obj)
```

试玩这个游戏

为了弄明白程序是如何运行，你可以像这样试玩：

```
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from ex48.parser import *
>>> x = parse_sentence([('verb', 'run'), ('direction', 'north')])
>>> x.subject
'player'
>>> x.verb
'run'
>>> x.object
'north'
>>> x = parse_sentence([('noun', 'bear'), ('verb', 'eat'), ('stop', 'the'), ('noun', '
honey')])
>>> x.subject
'bear'
>>> x.verb
'eat'
>>> x.object
'honey'
```

你应该测试的东西

为《习题 49》写一个完整的测试方案，确认代码中所有的东西都能正常工作，把测试代码放到文件 `tests/parser_tests.py` 中，测试代码中也要包含对异常的测试——输入一个错误的句子它会抛出一个异常来。

使用 `assert_raises` 这个函数来检查异常，在 `nose` 的文档里查看相关的内容，学着使用它写针对“执行失败”的测试，这也是测试很重要的一个方面。从 `nose` 文档中学会 `assert_raises` 以及一些别的函数的使用方法。

写完测试以后，你应该就明白了这段程序的工作原理，而且也学会了如何为别人的程序写测试代码。 相信我，这是一个非常有用的技能。

附加题

> 1. 修改 `parse_` 函数（方法），将它们放到一个类里边，而不仅仅是独立的方法函数。这两种程序设计你喜欢哪一种呢？
> 2. 提高 `parser` 的容错能力，这样即使用户输入了你预定义语汇之外的词语，你的程序也能正常运行下去。
> 3. 改进语法，让它可以处理更多的东西，例如数字。
> 4. 想想在游戏里你的 `Sentence` 类可以对用户输入做哪些有趣的事情。

常见问题

Q: 我好像不能让 `assert_raises` 正常运行

> 确认你写的是`assert_raises(exception, callable, parameters)`而不是`assert_raises(exception, callable(parameters))`。注意一下第二种写法中，调用了函数`callable`并将返回值传递给`assert_raises`，这种写法是错误的，你应该把要调用的函数也作为参数传递给`assert_raises`。

练习50.你的第一个网站

最后的3个练习将会很难，你需要在他们身上多花一些时间。第一个练习里你将创建一个简单的web版本的游戏。在你开始这节练习以前，你必须已经成功地完成过了《习题 46》的内容，正确安装了pip，而且学会了如何安装软件包以及如何创建项目骨架。如果你不记得这些内容，就回到《习题 46》重新复习一遍。

安装 lpthw.web

在创建你的第一个网页应用程序之前，你需要安装一个“Web 框架”，它的名字叫 `lpthw.web` 。所谓的“框架”通常是指“让某件事情做起来更容易的软件包”。在网页应用的世界里，人们创建了各种各样的“网页框架”，用来解决他们在创建网站时碰到的问题，然后把这些解决方案用软件包的方式发布出来，这样你就可以利用它们引导创建你自己的项目了。

可选的框架类型有很多很多，不过在这里我们将使用 `lpthw.web` 框架。你可以先学会它，等到差不多的时候再去接触其它的框架，不过 `lpthw.web` 本身挺不错的，所以就算你一直使用也没关系。

使用 `pip` 安装 `lpthw.web` ：

```
$ sudo pip install lpthw.web
[sudo] password for zedshaw:
Downloading/unpacking lpthw.web
  Running setup.py egg_info for package lpthw.web

Installing collected packages: lpthw.web
  Running setup.py install for lpthw.web

Successfully installed lpthw.web
Cleaning up...
```

以上是 Linux 和 Mac OSX 系统下的安装命令，如果你使用的是 Windows，那你只要把 `sudo` 去掉就可以了。如果你无法正常安装，请回到《习题 46》，确认自己学会了里边的内容。

> **Warning:** 其他 Python 程序员会警告你说 `lpthw.web` 只是另外一个叫做 `web.py` 的 Web 框架的代码分支(fork)，而 `web.py` 又包含了太多的“魔法(magic)”在里边。如果他们这么说的话，你告诉他们 `Google App Engine` 最早用的就是 `web.py` ，但没有一个 Python 程序员抱怨过它里边包含了太多的魔法，因为 Google 用它也没啥问题。如果Google觉得它可以，那它对你来说也不会差。所以还是回去继续学习吧，他们这些说法与其说是教导你，不如说是拿他们自己的教条束缚你，你还是忽略这些说法好了。

写一个简单的“Hello World”项目

现在我们使用 `lpthw.web` 做一个非常简单的“Hello World”项目出来，首先你要创建一个项目目录：

```
$ cd projects
$ mkdir gothonweb
$ cd gothonweb
$ mkdir bin gothonweb tests docs templates
$ touch gothonweb/__init__.py
$ touch tests/__init__.py
```

你最终的目的是把《习题43》中的游戏做成一个web应用，所以你的项目名称叫做 `gothonweb` ，不过在此之前，你需要创建一个最基本的 `lpthw.web` 应用，将下面的代码放到 `bin/app.py` 中：

```
import web

urls = (
  '/', 'index'
)

app = web.application(urls, globals())

class index:
    def GET(self):
        greeting = "Hello World"
        return greeting

if __name__ == "__main__":
    app.run()
```

然后使用下面的方法来运行这个 web 程序：

```
$ python bin/app.py
http://0.0.0.0:8080/
```

如果你这样做：

```
$ cd bin/   # WRONG! WRONG! WRONG!
$ python app.py  # WRONG! WRONG! WRONG!
```

那么你就错了。在所有Python项目中，都不会用 `cd` 到下一级目录中去启动服务，你就在最顶层的目录启动服务，这样所有的系统可以访问所有的模块和文件。 去重读习题46并理解项目布局和如何使用它。

最后，使用你的网页浏览器，打开 URL `http://localhost:8080/` ，你应该看到两样东西，首先是浏览器里显示了 `Hello, world!` ，然后是你的命令行终端显示了如下的输出：

```
$ python bin/app.py
http://0.0.0.0:8080/
127.0.0.1:59542 - - [13/Jun/2011 11:44:43] "HTTP/1.1 GET /" - 200 OK
127.0.0.1:59542 - - [13/Jun/2011 11:44:43] "HTTP/1.1 GET /favicon.ico" - 404 Not Found
```

这些是 `lpthw.web` 打印出的 log 信息，从这些信息你可以看出服务器有在运行，而且能了解到程序在浏览器背后做了些什么事情。这些信息还有助于你发现程序的问题。例如在最后一行它告诉你浏览器试图获取 `/favicon.ico` ，但是这个文件并不存在，因此它返回的状态码是 `404 Not Found` 。

到这里，我还没有讲到任何 web 相关的工作原理，因为首先你需要完成准备工作，以便后面的学习能顺利进行，接下来的两节习题中会有详细的解释。我会要求你用各种方法把你的 lpthw.web 应用程序弄坏，然后再将其重新构建起来：这样做的目的是让你明白运行 lpthw.web 程序需要准备好哪些东西.

发生了什么？

在浏览器访问到你的网页应用程序时，发生了下面一些事情：

> 1. 浏览器通过网络连接到你自己的电脑，它的名字叫做 `localhost` ，这是一个标准称谓，表示的谁就是网络中你自己的这台计算机，不管它实际名字是什么，你都可以使用localhost 来访问。它使用到的网络端口是8080。
> 2. 连接成功以后，浏览器对 `bin/app.py` 这个应用程序发出了 HTTP 请求(request)，要求访问 URL `/` ，这通常是一个网站的第一个 URL。
> 3. 在 `bin/app.py` 里，我们有一个列表，里边包含了 URL 和类的匹配关系。我们这里只定义了一组匹配，那就是 `'/', 'index'` 的匹配。它的含义是：如果有人使用浏览器访问 / 这一级目录， `lpthw.web` 将找到并加载 `class index` ，从而用它处理这个浏览器请求。
> 4. 现在 `lpthw.web` 找到了 `class index` ，然后针对这个类的一个实例调用了 `index.GET` 这个方法函数。该函数运行后返回了一个字符串，以供 `lpthw.web` 将其传递给浏览器。
> 5. 最后 `lpthw.web` 完成了对于浏览器请求的处理，将响应(response)回传给浏览器，于是你就看到了现在的页面。

确定你真的弄懂了这些，你需要画一个流程图，来理清信息是如何从浏览器传递到 `lpthw.web` ，再到 `index.GET` ，再回到你的浏览器的。

修正错误

第一步，把第 11 行的 `greeting` 变量赋值删掉，然后刷新浏览器。你应该会看到一个错误页面，你可以通过这一页丰富的错误信息看出你的程序崩溃的原因是什么。当然你已经知道出错的原因是 `greeting` 的赋值丢失了，不过 `lpthw.web` 还是会给你一个挺好的错误页面，让你能找到出错的具体位置。试试在这个错误页面上做以下操作：

> 1. 检查每一段 `Local vars` 输出（用鼠标点击它们），追踪里边提到的变量名称，以及它们是在哪些代码文件中用到的。
> 2. 阅读 `Request Information` 一节，看看里边哪些知识是你已经熟悉了的。Request 是浏览器发给你的 `gothonweb` 应用程序的信息。这些知识对于日常网页浏览没有什么用处，但现在你要学会这些东西，以便写出 web 应用程序来。3.试着把这个小程序的别的位置改错，探索一下会发生什么事情。 `lpthw.web` 的会把一些错误信息和堆栈跟踪(stack trace)信息显示在命令行终端，所以别忘了检查命令行终端的信息输出。

创建基本的模板文件

你已经试过用各种方法把这个 lpthw.web 程序改错，不过你有没有注意到"Hello World"不是一个好 HTML 网页呢？这是一个 web 应用，所以需要一个合适的 HTML 响应页面才对。为了达到这个目的，下一步你要做的是将"Hello World"以较大的绿色字体显示出来。

第一步是创建一个 `templates/index.html` 文件，内容如下：

```
$def with (greeting)

<html>
    <head>
        <title>Gothons Of Planet Percal #25</title>
    </head>
<body>

$if greeting:
    I just wanted to say <em style="color: green; font-size: 2em;">$greeting</em>.
$else:
    <em>Hello</em>, world!

</body>
</html>
```

如果你学过 HTML 的话，这些内容你看上去应该很熟悉。如果你没学过 HTML，那你应该去研究一下，试着用 HTML 写几个网页，从而知道它的工作原理。不过我们这里的 HTML 文件其实是一个"模板(template)"，如果你向模板提供一些参数， `lpthw.web` 就会在模板中找到对应的位置，将参数的内容填充到模板中。例如每一个出现 `$greeting` 的位置， `$greeting` 的内容都会被替换成对应这个变量名的参数。

为了让你的 `bin/app.py` 处理模板，你需要写一写代码，告诉 `lpthw.web` 到哪里去找到模板进行加载，以及如何渲染(render)这个模板，按下面的方式修改你的 app.py：

```
import web

urls = (
  '/', 'Index'
)

app = web.application(urls, globals())

render = web.template.render('templates/')

class Index(object):
    def GET(self):
        greeting = "Hello World"
        return render.index(greeting = greeting)

if __name__ == "__main__":
    app.run()
```

特别注意一下 `render` 这个新变量名，注意我修改了 `index.GET` 的最后一行，让它返回了 `render.index()` ，并且将 `greeting` 变量作为参数传递给了这个函数。

改好上面的代码后，刷新一下浏览器中的网页，你应该会看到一条和之前不同的绿色信息输出。你还可以在浏览器中通过“查看源文件(View Source)”看到模板被渲染成了标准有效的 HTML 源代码。

这么讲也许有些太快了，我来详细解释一下模板的工作原理吧：

> 1. 在 `bin/app.py` 里面你添加了一个叫做 `render` 的新变量，它本身是一个 `web.template.render` 对象。
> 2. 你将 `templates/` 作为参数传递给了这个对象，这样就让 `render` 知道了从哪里去加载模板文件。
> 3. 在你后面的代码中，当浏览器一如既往地触发了 `index.GET` 以后，它没有再返回简单的 `greeting` 字符串，取而代之的是你调用了 `render.index` ，而且将问候语句作为一个变量传递给它。
> 4. 这个 `render_template` 函数可以说是一个“魔法函数”，它看到了你需要的是 `index.html` ，于是就跑到 `templates/` 目录下，找到名字为 `index.html` 的文件，然后就把它渲染(render)一遍（叫“转换一遍”也可以）。
> 5. 在 `templates/index.html` 文件中，你可以看到初始定义一行中说这个模板需要使用一个叫 `greeting` 的参数，这和函数定义中的格式差不多。另外和 Python 语法一样，模板文件是缩进敏感的，所以要确认自己弄对了缩进。
> 6. 最后，你让 `templates/index.html` 去检查 `greeting` 这个变量，如果这个变量存在的话，就打印出变量的内容，如果不存在的话，就会打印出一个默认的问候信息。

要深入理解这个过程，你可以修改 greeting 变量以及 HTML 模板的内容，看看会有什么效果。然后创建一个叫做 `templates/foo.html` 的模板，并且使用一个新的 `render.foo()` 去渲染它。从这个过程你也可以看出， `render` 调用的函数名称只要跟 `templates/` 下的 `.html` 文件名匹配到，这个 HTML 模板就可以被渲染到了。

附加题

> 1. 阅读 `http://webpy.org/` 里边的文档，它其实和 `lpthw.web` 是同一个项目。
> 2. 实验一下你在上述网站看到的所有的东西，包括里边的代码示例。
> 3. 阅读以下 HTML5 和 CSS3 相关的东西，自己练习着写几个 `.html` 和 `.css 文件` 。
> 4. 如果你有一个懂 Django 朋友可以帮你的话，你可以试着使用 Django 完成一下习题 50、51、52，看看结果会是什么样子的。

常见问题

Q: 我好想无法连接到 `http://localhost:8080/`

> 那么试试访问 `http://127.0.0.1:8080/`

Q: `lpthw.web` 和 `web.py` 有什么区别？

> 没有区别。我只是在特定版本“锁定” `web.py` ，以使它对所有学生都是一样的，然后再命名为 `lpthw.web` ，上一个版本的 `web.py` 可能就不同于这一版本。

Q: 我的代码找不到 `index.html` (或者其他文件)

> 你可能是先执行了 `cd bin/` ，不要执行这一句，所有的命令都应该在 `bin/` 的上一级目录执行，所以如果你不能执行 `python bin/app.py` ,说明你在错误的目录上。

Q: 当我们调用模板的时候，为什么要执行 `greeting=greeting` 赋值操作

> 你并没有给 `greeting` 赋值，你只是给模板设定一个命名参数。这是声明的一种，但它只影响调用模板的功能。

Q: 我的电脑上不能使用8080端口

> 你可能有一个杀毒程序占用了这个端口，试试别的端口。

Q: 安装 `lpthw.web` 时，我遇到报错信息 `ImportError "No module named web"`

> 你可能安装了多个版本的Python并且正在使用一个错误的版本，或者你是因为使用了一个旧版本的 `pip` ，导致安装没有成功，试着先卸载 `lpthw.web` ，在重装一次，如果还没有解决问题，再次确认下你是否使用了正确的Python版本。

练习51.从浏览器获取输入

虽然能让浏览器显示“Hello World”是很有趣的一件事情，但是如果能让用户通过表单(form)向你的应用程序提交文本就更有趣了。这节习题中，我们将使用 form 改进你的 web 程序，并且将用户相关的信息保存到他们的“会话(session)”中。

Web 的工作原理

该学点无趣的东西了。在创建 form 前你需要先多学一点关于 web的工作原理。这里讲的并不完整，但是相当准确，在你的程序出错时，它会帮你找到出错的原因。另外，如果你理解了 form 的应用，那么创建 form 对你来说就会更容易了。

我将以一个简单的图示讲起，它向你展示了 web 请求的各个不同的部分，以及信息传递的大致流程：

为了方便讲述 HTTP 请求(request) 的流程，我在每条线上面加了字母标签以作区别：

> 1. 你在浏览器中输入网址`http://test.com//`，然后浏览器会通过你的电脑的网络设备发出 request（线路 A）。
> 2. 你的 request 被传送到互联网（线路B），然后再抵达远端服务器（线路 C），然后我的服务器将接受这个 request。
> 3. 我的服务器接受 request 后，我的web应用程序就去处理这个请求（线路 D），然后我的 Python代码就会去运行`index.GET`这个“处理程序(handler)”。
> 4. 在代码`return`的时候，我的 Python 服务器就会发出响应(response)，这个响应会再通过线路 D 传递到你的浏览器。
> 5. 这个网站所在的服务器将响应由线路 D 获取，然后通过线路 C 传至互联网。
> 6. 响应通过互联网由线路 B 传至你的计算机，计算机的网卡再通过线路 A 将响应传给你的浏览器。
> 7. 最后，你的浏览器显示了这个响应的内容。

这段解释中用到了一些术语。你需要掌握这些术语，以便在谈论你的 web 应用时你能明白而且应用它们：

浏览器(browser)这是你几乎每天都会用到的软件。大部分人不知道它眞正的原理，他们只会把它叫作“the internet”。它的作用其实是接收你输入到地址栏网址(例如 `http://test.com/`)，然后使用该信息向该网址对应的服务器提出请求(request)。

地址(address)通常这是一个像`http://test.com/`一样的 URL (Uniform Resource Locator，统一资源定位器)，它告诉浏览器该打开哪个网站。前面的 `http` 指出了你要使用的协议(protocol)，这里我们用的是“超文本传输协议(Hyper-Text Transport Protocol)”。你还可以试试`ftp://ibiblio.org/` ，这是一个“FTP 文件传输协议(File Transport Protocol)”的例

子。`test.com` 这部分是“主机名(hostname)”，也就是一个便于人阅读和记忆的字串，主机名会被匹配到一串叫作“IP 地址”的数字上面，这个“IP 地址”就相当于网络中一台计算机的电话号码，通过这个号码可以访问到这台计算机。最后，URL 中还可以尾随一个“路径”，例如 `http://test.com//book/` 中的 `/book/` ，它对应的是服务器上的某个文件或者某些资源，通过访问这样的网址，你可以向服务器发出请求，然后获得这些资源。网站地址还有很多别的组成部分，不过这些是最主要的。

连接(connection)一旦浏览器知道了协议(http)、服务器(`http://test.com/`)、以及要获得的资源，它就要去创建一个连接。这个过程中，浏览器让操作系统(Operating System, OS)打开计算机的一个“端口(port)”（通常是 80 端口），端口准备好以后，操作系统会回传给你的程序一个类似文件的东西，它所做的事情就是通过网络传输和接收数据，让你的计算机和 `http://test.com/` 这个网站所属的服务器之间实现数据交流。 当你使用 `http://localhost:8080/` 访问你自己的站点时，发生的事情其实是一样的，只不过这次你告诉了浏览器要访问的是你自己的计算机(localhost)，要使用的端口 不是默认的 80，而是 8080。你还可以直接访问 `http://test.com:80/` ， 这和不输入端口效果一样，因为 HTTP 的默认端口本来就是 80。

请求(request)你的浏览器通过你提供的地址建立了连接，现在它需要从远端服务器要到它（或你）想要的资源。如果你在 URL 的结尾加了 `/book/` ，那你想要的就是 `/book/` 对应的文件或资源，大部分的服务器会直接为你调用 `/book/index.html` 这个文件，不过我们就假装不存在好了。浏览器为了获得服务器上的资源，它需要向服务器发送一个“请求”。这里我就不讲细节了，为了得到服务器上的内容，你必须先向服务器发送一个请求才行。有意思的是，“资源”不一定非要是文件。例如当浏览器向你的应用程序提出请求的时候，服务器返回的其实是你的 Python 代码生成的一些东西。

服务器(server)服务器指的是浏览器另一端连接的计算机，它知道如何回应浏览器请求的文件和资源。大部分的 web 服务器只要发送文件就可以了，这也是服务器流量的主要部分。不过你学的是使用 Python 组建一个服务器，这个服务器知道如何接受请求，然后返回用 Python 处理过的字符串。当你使用这种处理方式时，你其实是假装把文件发给了浏览器，其实你用的都只是代码而已。就像你在《习题50》中看到的，要构建一个“响应”其实也不需要多少代码。

响应(response)这就是你的服务器回复你的请求，发回至浏览器的 HTML，它里边可能有 css、javascript、或者图像等内容。以文件响应为例，服务器只要从磁盘读取文件，发送给浏览器就可以了，不过它还要将这些内容包在一个特别定义的“头部信息(header)”中，这样浏览器就会知道它获取的是什么类型的内容。以你的 web 应用程序为例，你发送的其实还是一样的东西，包括 header 也一样，只不过这些数据是你用 Python 代码即时生成的。

这个可能是你能在网上找到的关于浏览器如何访问网站的最快的快速课程了。这节课程应该可以帮你更容易地理解本节的习题，如果你还是不明白，就到处找资料多多了解这方面的信息，直到你明白为止。有一个很好的方法，就是你对照着上面的图示，将你在《习题 50》中创建的 web 程序中的内容分成几个部分，让其中的各部分对应到上面的图示。如果你可以正确地将程序的各部分对应到这个图示，你就大致明白它的工作原理了。

表单(form) 的工作原理

熟悉“表单”最好的方法就是写一个可以接收表单数据的程序出来，然后看你可以对它做些什么。先将你的 `bin/app.py` 修改成下面的样子：

```
import web

urls = (
  '/hello', 'Index'
)

app = web.application(urls, globals())

render = web.template.render('templates/')

class Index(object):
    def GET(self):
        form = web.input(name="Nobody")
        greeting = "Hello, %s" % form.name

        return render.index(greeting = greeting)

if __name__ == "__main__":
    app.run()
```

重启你的 web 程序（按 CTRL-C 后重新运行），确认它有运行起来，然后使用浏览器访问 `http://localhost:8080/hello` ，这时浏览器应该会显示“I just wanted to say Hello, Nobody.”，接下来，将浏览器的地址改成 `http://localhost:8080/hello?name=Frank` ，然后你可以看到页面显示为“Hello, Frank.”，最后将 `name=Frank` 修改为你自己的名字，你就可以看到它对你说“Hello”了。

让我们研究一下你的程序里做过的修改:

> 1. 我们没有直接为 `greeting` 赋值，而是使用了 `web.input` 从浏览器获取数据。这个函数会将一组 key=value 的表述作为默认参数，解析你提供的 URL 中的 `?name=Frank` 部分，然后返回一个对象，你可以通过这个对象方便地访问到表单的值。
> 2. 然后我通过 `form` 对象的 `form.name` 属性为 `greeting` 赋值，这句你应该已经熟悉了。
> 3. 其他的内容和以前是一样的。

URL 中该还可以包含多个参数。将本例的 URL 改成这样子：`http://localhost:8080/hello?name=Frank&greet=Hola` 。然后修改代码，让它去获取 `form.name` 和 `form.greet` ，如下所示：

```
greeting = "%s, %s" % (form.greet, form.name)
```

修改完毕后，试着访问新的 URL。然后将 `&greet=Hola` 部分删除，看看你会得到什么样的错误信息。由于我们在 `web.input(name="Nobody")` 中没有为 `greet` 设定默认值，这样 `greet` 就变成了一个必须的参数，如果没有这个参数程序就会报错。现在修改一下你的程序，在

`web.input` 中为 `greet` 设一个默认值试试看。另外你还可以设 greet=None，这样你可以通过程序检查 `greet` 的值是否存在，然后提供一个比较好的错误信息出来，例如：

```
form = web.input(name="Nobody", greet=None)

if form.greet:
    greeting = "%s, %s" % (form.greet, form.name)
    return render.index(greeting = greeting)
else:
    return "ERROR: greet is required."
```

创建 HTML 表单

你可以通过 URL 参数实现表单提交，不过这样看上去有些丑陋，而且不方便一般人使用，你真正需要的是一个“POST 表单”，这是一种包含了 `<form>` 标签的特殊 HTML 文件。这种表单收集用户输入并将其传递给你的 web 程序，这和你上面实现的目的基本是一样的。

让我们来快速创建一个，从中你可以看出它的工作原理。你需要创建一个新的 HTML 文件，叫做 `templates/hello_form.html` ：

```
<html>
    <head>
        <title>Sample Web Form</title>
    </head>
<body>

<h1>Fill Out This Form</h1>

<form action="/hello" method="POST">
    A Greeting: <input type="text" name="greet">
    <br/>
    Your Name: <input type="text" name="name">
    <br/>
    <input type="submit">
</form>

</body>
</html>
```

然后修改 `bin/app.py` :

```
import web

urls = (
  '/hello', 'Index'
)

app = web.application(urls, globals())

render = web.template.render('templates/')

class Index(object):
    def GET(self):
        return render.hello_form()

    def POST(self):
        form = web.input(name="Nobody", greet="Hello")
        greeting = "%s, %s" % (form.greet, form.name)
        return render.index(greeting = greeting)

if __name__ == "__main__":
    app.run()
```

都写好以后，重启 web 程序，然后通过你的浏览器访问它。

这回你会看到一个表单，它要求你输入“一个问候语句(A Greeting)”和“你的名字(Your Name)”，等你输入完后点击“提交(Submit)”按钮，它就会输出一个正常的问候页面，不过这一次你的URL 还是 `http://localhost:8080/hello` ，并没有添加参数进去。

在 `hello_form.html` 里面关键的一行是 `<form action="/hello" method="POST">` ，它告诉你的浏览器以下内容：

> 1. 从表单中的各个栏位收集用户输入的数据。
> 2. 让浏览器使用一种 `POST` 类型的请求，将这些数据发送给服务器。这是另外一种浏览器请求，它会将表单栏位“隐藏”起来。
> 3. 将这个请求发送至 `/hello` URL(这是由 `action="/hello"` 告诉浏览器的)。

你可以看到两段 `<input>` 标签的名字属性(name)和代码中的变量是对应的，另外我们在 `class index` 中使用的不再只是 `GET` 方法，而是另一个 `POST` 方法。

这个新程序的工作原理如下：

> 1. 浏览器访问到 web 程序的 `/hello` 目录，它发送了一个 `GET` 请求，于是我们的 `index.GET` 函数就运行并返回了 `hello_form` 。
> 2. 你填好了浏览器的表单，然后浏览器依照 `<form>` 中的要求，将数据通过 `POST` 请求的方式发给 web 程序。
> 3. Web 程序运行了 `index.POST` 方法（不是 `index.GET` 方法）来处理这个请求。
> 4. 这个 `index.POST` 方法完成了它正常的功能，将 hello 页面返回，这里并没有新的东西，只是一个新函数名称而已。

作为练习，在 `templates/index.html` 中添加一个链接，让它指向 `/hello` ，这样你可以反复填写并提交表单查看结果。确认你可以解释清楚这个链接的工作原理，以及它是如何让你实现在 `templates/index.html` 和 `templates/hello_form.html` 之间循环跳转的，还有就是要明

白你新修改过的 Python 代码，你需要知道在什么情况下会运行到哪一部分代码。

创建布局模板(layout template)

在你下一节练习创建游戏的过程中，你需要创建很多的小 HTML 页面。如果你每次都写一个完整的网页，你会很快感觉到厌烦的。幸运的 是你可以创建一个“布局模板”，也就是一种提供了通用的头文件和脚注的外壳模板，你可以用它将你所有的其他网页包裹起来。好程序员会尽可能减少重复动作，所以要做一个好程序员，使用布局模板是很重要的。

将 `templates/index.html` 修改成这样：

```
$def with (greeting)

$if greeting:
    I just wanted to say <em style="color: green; font-size: 2em;">$greeting</em>.
$else:
    <em>Hello</em>, world!
```

然后修改 `templates/hello_form.html`:

```
<h1>Fill Out This Form</h1>

<form action="/hello" method="POST">
    A Greeting: <input type="text" name="greet">
    <br/>
    Your Name: <input type="text" name="name">
    <br/>
    <input type="submit">
</form>
```

上面这些修改的目的，是将每一个页面顶部和底部的反复用到的“boilerplate”代码剥掉。这些被剥掉的代码会被放到一个单独的 `templates/layout.html` 文件中，从此以后，这些反复用到的代码就由 `layout.html` 来提供了。

上面的都改好以后，创建一个 `templates/layout.html` 文件，内容如下：

```
$def with (content)

<html>
<head>
    <title>Gothons From Planet Percal #25</title>
</head>
<body>

$:content

</body>
</html>
```

这个文件和普通的模板文件类似，不过其它的模板的内容将被传递给它，然后它会将其它 模板的内容“包裹”起来。任何写在这里的内容多无需写在别的模板中了。你需要注意`$:content`的用法，这和其它的模板变量有些不同。

最后一步，就是将`render`对象改成这样：render = web.template.render('templates/', base="layout")

这会告诉`lpthw.web`让它去使用`templates/layout.html`作为其它模板的基础模板。重启你的程序观察一下，然后试着用各种方法修改你的 layout 模板，不要修改你别的模板，看看输出会有什么样的变化。

为表单撰写自动测试代码

使用浏览器测试 web 程序是很容易的，只要点刷新按钮就可以了。不过毕竟我们是程序员嘛，如果我们可以写一些代码来测试我们的程序，为什么还要重复手动测试呢？接下来你要做的，就是为你的 web 程序写一个小测试。这会用到你在《习题 47》学过的一些东西，如果你不记得的话，可以复习一下。

为了让 Python 加载`bin/app.py`并进行测试，你需要先做一点准备工作。首先创建一个`bin/__init__.py`空文件，这样 Python 就会将`bin/`当作一个目录了。（在《习题 52》中你会去修改`__init__.py`，不过这是后话。）

我还为`lpthw.web`创建了一个简单的小函数，让你判断(assert) web 程序的响应，这个函数的名字，叫`assert_response`。创建一个`tests/tools.py`文件，内容如下：

```
from nose.tools import *
import re

def assert_response(resp, contains=None, matches=None, headers=None, status="200"):

    assert status in resp.status, "Expected response %r not in %r" % (status, resp.sta
tus)

    if status == "200":
        assert resp.data, "Response data is empty."

    if contains:
        assert contains in resp.data, "Response does not contain %r" % contains

    if matches:
        reg = re.compile(matches)
        assert reg.matches(resp.data), "Response does not match %r" % matches

    if headers:
        assert_equal(resp.headers, headers)
```

准备好这个文件以后，你就可以为你的`bin/app.py`写自动测试代码了。创建一个新文件，叫做`tests/app_tests.py`，内容如下：

```
from nose.tools import *
from bin.app import app
from tests.tools import assert_response

def test_index():
    # check that we get a 404 on the / URL
    resp = app.request("/")
    assert_response(resp, status="404")

    # test our first GET request to /hello
    resp = app.request("/hello")
    assert_response(resp)

    # make sure default values work for the form
    resp = app.request("/hello", method="POST")
    assert_response(resp, contains="Nobody")

    # test that we get expected values
    data = {'name': 'Zed', 'greet': 'Hola'}
    resp = app.request("/hello", method="POST", data=data)
    assert_response(resp, contains="Zed")
```

最后，使用 `nosetests` 运行测试脚本，然后测试你的 web 程序。

```
$ nosetests
.
----------------------------------------------------------------------
Ran 1 test in 0.059s

OK
```

这里我所做的，是将 `bin/app.py` 这个模块中的整个 web 程序都 import 进来，然后手动运行这个 web 程序。 `lpthw.web` 有一个非常简单的 API 用来处理请求，看上去大致是这样子的：

```
app.request(localpart='/', method='GET', data=None, host='0.0.0.0:8080',
            headers=None, https=False)
```

你可以将 URL 作为第一个参数，然后你可以修改 request 的方法、form 的数据、以及 header 的内容，这样你无须启动 web 服务器，就可以使用自动测试来测试你的 web 程序了。

为了验证函数的响应，你需要使用 `tests.tools` 中定义的 `assert_response` 函数，用法属下：

```
assert_response(resp, contains=None, matches=None, headers=None, status="200")
```

把你调用 `app.request` 得到的响应传递给这个函数，然后将你要检查的内容作为参数传递给该这个函数。你可以使用 `contains` 参数来检查响应中是否包含指定的值，使用 `status` 参数可以检查指定的响应状态。这个小函数其实包含了很多的信息，所以你还是自己研究一下的比较好。

在 `tests/app_tests.py` 自动测试脚本中，我首先确认 `/` 返回了一个“404 Not Found”响应，因为这个 URL 其实是不存在的。然后我检查了 `/hello` 在 `GET` 和 `POST` 两种请求的情况下都能正常工作。就算你没有弄明白测试的原理，这些测试代码应该是很好读懂的。

花一些时间研究一下这个最新版的 web 程序，重点研究一下自动测试的工作原理。确认你理解了将 `bin/app.py` 做为一个模块导入，然后进行自动化测试的流程。这是一个很重要的技巧，它会引导你学到更多东西。

附加题

> 1. 阅读和 HTML 相关的更多资料，然后为你的表单设计一个更好的输出格式。你可以先在纸上设计出来，然后用 HTML 去实现它。
> 2. 这是一道难题，试着研究一下如何进行文件上传，通过网页上传一张图像，然后将其保存到磁盘中。
> 3. 更难的难题，找到 `HTTP RFC` 文件（讲述 HTTP 工作原理的技术文件），然后努力阅读一下。这是一篇很无趣的文档，不过偶尔你会用到里边的一些知识。
> 4. 又是一道难题，找人帮你设置一个 web 服务器，例如 `Apache` 、 `Nginx` 、或者 `thttpd` 。试着让服务器服务一下你创建的 .html 和 .css 文件。如果失败了也没关系，web 服务器本来就都有点挫。
> 5. 完成上面的任务后休息一下，然后试着多创建一些 web 程序出来。你应该仔细阅读 `web.py` (它和 `lpthw.web` 是同一个程序)中关于会话(session)的内容，这样你可以明白如何保持用户的状态信息。

常见问题

Q: 我遇到报错信息 `ImportError "No module named bin.app"`

> 这个问题要么是你在错误的目录下启动了服务，要么是没有 `bin/__init__.py` 这个文件，再或者是你没有在你的shell中设置 `PYTHONPATH=.` 。请永远记住这些解决方案，因为这些错误是如此令人难以置信的普遍发生，当他们发生的时候，还会拖慢你服务的速度。

Q: 当我运行模板的时候，我遇到报错 `__template__() takes no arguments (1 given)`

> 你可能忘记把这个变量 `$def with (greeting)` 或者类似的变量放在模板的顶部了。

练习52.开始你的web游戏

这本书马上就要结束了。本章的练习对你是一个真正的挑战。当你完成以后，你就可以算是一个能力不错的Python初学者了。为了进一步学习，你还需要多读一些书，多写一些程序，不过你已经具备进一步学习的技能了。接下来的学习就只是时间、动力、以及资源的问题了。

在本章习题中，我们不会去创建一个完整的游戏，取而代之的是我们会为《习题 47》中的游戏创建一个"引擎(engine)"，让这个游戏能够在浏览器中运行起来。这会涉及到将《习题 43》中的游戏"重构(refactor)"，将《习题 47》中的架构混合进来，添加自动测试代码，最后创建一个可以运行游戏的 web 引擎。

这是一节很庞大的习题。我预测你要花一周到一个月才能完成它。最好的方法是一点点来，每天晚上完成一点，在进行下一步之前确认上一步有正确完成。

重构习题43的游戏

你已经在两个练习中修改了 `gothonweb` 项目，这节习题中你会再修改一次。这种修改的技术叫做"重构(refactoring)"，或者用我喜欢的讲法来说，叫"修修补补(fixing stuff)"。重构是一个编程术语，它指的是清理旧代码或者为旧代码添加新功能的过程。你其实已经做过这样的事情了，只不过不知道这个术语而已。这是写软件过程的第二个自然属性。

你在本节中要做的，是将《习题 47》中的可以测试的房间地图，以及《习题 43》中的游戏这两样东西归并到一起，创建一个新的游戏架构。游戏的内容不会变化，只不过我们会通过"重构"让它有一个更好的架构而已。

第一步是将 `ex47/game.py` 的内容复制到 `gothonweb/map.py` 中，然后将 `tests/ex47_tests.py` 的内容复制到 `tests/map_tests.py` 中，然后再次运行 `nosetests` ，确认他们还能正常工作。

> **NOTE:**从现在开始我不会再向你展示运行测试的输出了，我就假设你回去运行这些测试，而且知道怎样的输出是正确的。

将《习题 47》的代码拷贝完毕后，你就该开始重构它，让它包含《习题43》中的地图。我一开始会把基本架构为你准备好，然后你需要去完成 `map.py` 和 `map_tests.py` 里边的内容。

首先要做的是使用 `Room` 类来构建基本的地图架构：

```
class Room(object):

    def __init__(self, name, description):
        self.name = name
        self.description = description
        self.paths = {}
```

```
    def go(self, direction):
        return self.paths.get(direction, None)

    def add_paths(self, paths):
        self.paths.update(paths)

central_corridor = Room("Central Corridor",
"""
The Gothons of Planet Percal #25 have invaded your ship and destroyed
your entire crew.  You are the last surviving member and your last
mission is to get the neutron destruct bomb from the Weapons Armory,
put it in the bridge, and blow the ship up after getting into an
escape pod.

You're running down the central corridor to the Weapons Armory when
a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume
flowing around his hate filled body.  He's blocking the door to the
Armory and about to pull a weapon to blast you.
""")

laser_weapon_armory = Room("Laser Weapon Armory",
"""
Lucky for you they made you learn Gothon insults in the academy.
You tell the one Gothon joke you know:
Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr.
The Gothon stops, tries not to laugh, then busts out laughing and can't move.
While he's laughing you run up and shoot him square in the head
putting him down, then jump through the Weapon Armory door.

You do a dive roll into the Weapon Armory, crouch and scan the room
for more Gothons that might be hiding.  It's dead quiet, too quiet.
You stand up and run to the far side of the room and find the
neutron bomb in its container.  There's a keypad lock on the box
and you need the code to get the bomb out.  If you get the code
wrong 10 times then the lock closes forever and you can't
get the bomb.  The code is 3 digits.
""")

the_bridge = Room("The Bridge",
"""
The container clicks open and the seal breaks, letting gas out.
You grab the neutron bomb and run as fast as you can to the
bridge where you must place it in the right spot.

You burst onto the Bridge with the netron destruct bomb
under your arm and surprise 5 Gothons who are trying to
take control of the ship.  Each of them has an even uglier
clown costume than the last.  They haven't pulled their
weapons out yet, as they see the active bomb under your
arm and don't want to set it off.
""")

escape_pod = Room("Escape Pod",
"""
You point your blaster at the bomb under your arm
and the Gothons put their hands up and start to sweat.
You inch backward to the door, open it, and then carefully
place the bomb on the floor, pointing your blaster at it.
You then jump back through the door, punch the close button
and blast the lock so the Gothons can't get out.
Now that the bomb is placed you run to the escape pod to
get off this tin can.

You rush through the ship desperately trying to make it to
the escape pod before the whole ship explodes.  It seems like
hardly any Gothons are on the ship, so your run is clear of
interference.  You get to the chamber with the escape pods, and
now need to pick one to take.  Some of them could be damaged
but you don't have time to look.  There's 5 pods, which one
do you take?
""")
```

```
the_end_winner = Room("The End",
"""
You jump into pod 2 and hit the eject button.
The pod easily slides out into space heading to
the planet below.  As it flies to the planet, you look
back and see your ship implode then explode like a
bright star, taking out the Gothon ship at the same
time.  You won!
""")

the_end_loser = Room("The End",
"""
You jump into a random pod and hit the eject button.
The pod escapes out into the void of space, then
implodes as the hull ruptures, crushing your body
into jam jelly.
"""
)

escape_pod.add_paths({
    '2': the_end_winner,
    '*': the_end_loser
})

generic_death = Room("death", "You died.")

the_bridge.add_paths({
    'throw the bomb': generic_death,
    'slowly place the bomb': escape_pod
})

laser_weapon_armory.add_paths({
    '0132': the_bridge,
    '*': generic_death
})

central_corridor.add_paths({
    'shoot!': generic_death,
    'dodge!': generic_death,
    'tell a joke': laser_weapon_armory
})

START = central_corridor
```

你会发现我们的 `Room` 类和地图有一些问题：

> 1. 在进入一个房间以前会打印出一段文字作为房间的描述，我们需要将这些描述和每个房间关联起来，这样房间的次序就不会被打乱了，这对我们的游戏是一件好事。这些描述本来是在 `if-else` 结构中的，这是我们后面要修改的东西。
> 2. 原版游戏中我们使用了专门的代码来生成一些内容，例如炸弹的激活键码，舰舱的选择等，这次我们做游戏时就先使用默认值好了，不过后面的附加题里，我会要求你把这些功能再加到游戏中。
> 3. 我为所有的游戏中的失败结尾写了一个 `generic_death` ，你需要去补全这个函数。你需要把原版游戏中所有的失败结尾都加进去，并确保代码能正确运行。
> 4. 我添加了一种新的转换模式，以"*"为标记，用来在游戏引擎中实现“catch-all”动作。

等你把上面的代码基本写好以后，接下来就是引导你继续写下去的自动测试的内容 `tests/map_test.py` 了：

```
from nose.tools import *
from gothonweb.map import *

def test_room():
    gold = Room("GoldRoom",
                """This room has gold in it you can grab. There's a
                door to the north.""")
    assert_equal(gold.name, "GoldRoom")
    assert_equal(gold.paths, {})

def test_room_paths():
    center = Room("Center", "Test room in the center.")
    north = Room("North", "Test room in the north.")
    south = Room("South", "Test room in the south.")

    center.add_paths({'north': north, 'south': south})
    assert_equal(center.go('north'), north)
    assert_equal(center.go('south'), south)

def test_map():
    start = Room("Start", "You can go west and down a hole.")
    west = Room("Trees", "There are trees here, you can go east.")
    down = Room("Dungeon", "It's dark down here, you can go up.")

    start.add_paths({'west': west, 'down': down})
    west.add_paths({'east': start})
    down.add_paths({'up': start})

    assert_equal(start.go('west'), west)
    assert_equal(start.go('west').go('east'), start)
    assert_equal(start.go('down').go('up'), start)

def test_gothon_game_map():
    assert_equal(START.go('shoot!'), generic_death)
    assert_equal(START.go('dodge!'), generic_death)

    room = START.go('tell a joke')
    assert_equal(room, laser_weapon_armory)
```

你在这部分练习中的任务是完成地图，并且让自动测试可以完整地检查过整个地图。这包括将所有的 `generic_death` 对象修正为游戏中实际的失败结尾。让你的代码成功运行起来，并让你的测试越全面越好。后面我们会对地图做一些修改，到时候这些测试将保证修改后的代码还可以正常工作。

会话(session)和用户跟踪

在你的 web 程序运行的某个位置，你需要追踪一些信息，并将这些信息和用户的浏览器关联起来。在HTTP协议的框架中，web环境是“无状态(stateless)”的，这意味着你的每一次请求都是独立于其他请求的。如果你请求了页面A，输入了一些数据，然后点了一个页面B的链接，那你在页面A输入的数据就全部消失了。

解决这个问题的方法是为 web 程序建立一个很小的数据存储功能，给每个浏览器进程赋予一个独一无二的数字，用来跟踪浏览器所作的事情。这个存储通常用数据库或者存储在磁盘上的文件来实现。这就是所谓的“会话跟踪”和在浏览器中使用Cookies以保持用户状态。
在 `lpthw.web` 这个小框架中实现这样的功能是很容易的，以下就是一个这样的例子：

```
import web

web.config.debug = False

urls = (
    "/count", "count",
    "/reset", "reset"
)
app = web.application(urls, locals())
store = web.session.DiskStore('sessions')
session = web.session.Session(app, store, initializer={'count': 0})

class count:
    def GET(self):
        session.count += 1
        return str(session.count)

class reset:
    def GET(self):
        session.kill()
        return ""

if __name__ == "__main__":
    app.run()
```

为了实现这个功能，你需要创建一个 `sessions/` 文件夹作为程序的会话存储位置，创建好以后运行这个程序，然后检查 `/count` 页面，刷新一下这个页面，看计数会不会累加上去。关掉浏览器后，程序就会“忘掉”之前的位置，这也是我们的游戏所需的功能。有一种方法可以让浏览器永远记住一些信息，不过这会让测试和开发变得更难。如果你回到 `/reset/` 页面，然后再访问 `/count` 页面，你可以看到你的计数器被重置了，因为你已经把会话杀掉了。

你需要花点时间弄懂这段代码，注意会话开始时 `count` 的值是如何设为 0 的。另外再看看 `sessions/` 下面的文件，看你能不能把它们打开。下面是我把一个 Python 会话打开并且解码的过程：

```
>>> import pickle
>>> import base64
>>> base64.b64decode(open("sessions/XXXXX").read())
"(dp1\nS'count'\np2\nI1\nsS'ip'\np3\nV127.0.0.1\np4\nsS'session_id'\np5\nS'XXXX'\np6\n
s."
>>>
>>> x = base64.b64decode(open("sessions/XXXXX").read())
>>>
>>> pickle.loads(x)
{'count': 1, 'ip': u'127.0.0.1', 'session_id': 'XXXXX'}
```

所以会话其实就是使用 `pickle` 和 `base64` 这些库写到磁盘上的字典。存储和管理会话的方法很多，大概和 Python 的 web 框架那么多，所以了解它们的工作原理并不重要。当然如果你需要调试或者清空会话时，知道点原理还是有用的。

创建引擎

你应该已经写好了游戏地图和它的单元测试代码。现在我要求你制作一个简单的游戏引擎，用来让游戏中的各个房间运转起来，从玩家收集输入，并且记住玩家到了那一幕。我们将用到你刚学过的会话来制作一个简单的引擎，让它可以：

> 1. 为新用户启动新的游戏。
> 2. 将房间展示给用户。
> 3. 接受用户的输入。
> 4. 在游戏中处理用户的输入。
> 5. 显示游戏的结果，继续游戏的下一幕，知道玩家角色死亡为止.

为了创建这个引擎，你需要将我们久经考验的 `bin/app.py` 搬过来，创建一个功能完备的、基于会话的游戏引擎。这里的难点是我会先使用基本的 HTML 文件创建一个非常简单的版本，接下来将由你完成它，基本的引擎是这个样子的：

```
import web
from gothonweb import map

urls = (
  '/game', 'GameEngine',
  '/', 'Index',
)

app = web.application(urls, globals())

# little hack so that debug mode works with sessions
if web.config.get('_session') is None:
    store = web.session.DiskStore('sessions')
    session = web.session.Session(app, store,
                                  initializer={'room': None})
    web.config._session = session
else:
    session = web.config._session

render = web.template.render('templates/', base="layout")

class Index(object):
    def GET(self):
        # this is used to "setup" the session with starting values
        session.room = map.START
        web.seeother("/game")

class GameEngine(object):

    def GET(self):
        if session.room:
            return render.show_room(room=session.room)
        else:
            # why is there here? do you need it?
            return render.you_died()

    def POST(self):
        form = web.input(action=None)

        # there is a bug here, can you fix it?
        if session.room and form.action:
            session.room = session.room.go(form.action)

        web.seeother("/game")

if __name__ == "__main__":
    app.run()
```

这个脚本里你可以看到更多的新东西，不过了不起的事情是，整个基于网页的游戏引擎只要一个小文件就可以做到了。这段脚本里最有技术含量的事情就是将会话带回来的那几行，这对于调试模式下的代码重载是必须的，否则每次你刷新网页，会话就会消失，游戏也不会再继续了。

在你运行 `bin/app.py` 之前，你需要修改 `PYTHONPATH` 环境变量。不知道什么是环境变量？为了运行一个最基本的 Python 程序，你就得学会环境变量，Python 的这一点确实有点挫。不过没办法，用 Python 的人就喜欢这样：在你的命令行终端，输入:

```
export PYTHONPATH=$PYTHONPATH:.
```

如果你用的是 Windows，那就输入:

```
$env:PYTHONPATH = "$env:PYTHONPATH;."
```

你只要针对每一个命令行会话界面输入一次就可以了，不过如果你运行 Python 代码时看到了import错误，那你就需要去执行一下上面的命令，或者也许是因为你上次执行的 有错才导致 import 错误的。

接下来你需要删掉 `templates/hello_form.html` 和 `templates/index.html` ，然后重新创建上面代码中提到的两个模板。这里是一个非常简单的 `templates/show_room.html` 供你参考：

```
$def with (room)

<h1> $room.name </h1>

<pre>
$room.description
</pre>

$if room.name == "death":
    <p><a href="/">Play Again?</a></p>
$else:
    <p>
    <form action="/game" method="POST">
        - <input type="text" name="action"> <input type="SUBMIT">
    </form>
    </p>
```

这就用来显示游戏中的房间的模板。接下来，你需要在用户跑到地图的边界时，用一个 模板告诉用户他的角色的死亡信息，也就是 `templates/you_died.html` 这个模板：

```
<h1>You Died!</h1>

<p>Looks like you bit the dust.</p>
<p><a href="/">Play Again</a></p>
```

准备好了这些文件，你现在可以做下面的事情了：

> 1. 让测试代码 `tests/app_tests.py` 再次运行起来，这样你就可以去测试这个游戏。由于会话的存在，你可能顶多只能实现几次点击，不过你应该可以做出一些基本的测试来。
> 2. 删除 `sessions/*` 下的文件，再重新运行一遍游戏，确认游戏是从一开始运行起来的。
> 3. 执行 `python bin/app.py` 脚本，试玩一下你的游戏。

你需要和往常一样刷新和修正你的游戏，慢慢修改游戏的 HTML 文件和引擎，直到你实现游戏需要的所有功能为止。

你的期末考试

你有没有觉着我一下子给了你超多的信息呢？那就对了，我想要你在学习技能的同时可以有一些可以用来鼓捣的东西。为了完成这节习题，我将给你最后一套需要你自己完成的练习。你将注意到，到目前为止你写的游戏并不是很好，这只是你的第一版代码而已。你现在的任务是通过做这些事让游戏更加完善：

> 1. 修正代码中所有我提到和没提到的 bug，如果你发现了新的 bug，你可以告诉我。
> 2. 改进所有的自动测试，让你可以测试更多的内容，直到你可以不用浏览器就能测到所有的内容为止。
> 3. 让 HTML 页面看上去更美观一些。
> 4. 研究一下网页登录系统，为这个程序创建一个登录界面，这样人们就可以登录这个游戏，并且可以保存游戏高分。
> 5. 完成游戏地图，尽可能地把游戏做大，功能做全。
> 6. 给用户一个“帮助系统”，让他们可以查询每个房间里可以执行哪些命令。
> 7. 为你的游戏添加新功能，想到什么功能就添加什么功能。
> 8. 创建多个地图，让用户可以选择他们想要玩的一张来进行游戏。你的 `bin/app.py` 应该可以运行提供给它的任意的地图，这样你的引擎就可以支持多个不同的游戏。
> 9. 最后，使用你在习题 48 和 49 中学到的东西来创建一个更好的输入处理器。你手头已经有了大部分必要的代码，你只需要改进语法，让它和你的输入表单以及游戏引擎挂钩即可。

祝你好运！

常见问题

Q: 我在游戏中使用了 `sessions` ，但是我没办法利用 `nosetests` 来测试它

> 你需要了解sessions的机制。`http://webpy.org/cookbook/session_with_reloader` 。

Q: 我遇到报错 `ImportError`

> 错误的目录。错误的Python版本。 `PYTHONPATH` 没有设置，没有 `__init__.py` 文件，检查 `import` 中的拼写错误。

来自老程序员的建议

你已经完成了这本书而且打算继续编程。也许这会成为你的一门职业，也许你只是作为业余爱好玩玩。无论如何，你都需要一些建议以保证你在正确的道路上继续前行，并且让这项新的爱好为你带来最大程度的享受。

我从事编程已经太长时间，长到对我来说编程已经是非常乏味的事情了。我写这本书的时候，已经懂得大约20种编程语言，而且可以在大约一天或者一个星期内学会一门编程语言(取决于这门语言有多古怪)。现在对我来说编程这件事情已经很无聊，已经谈不上什么兴趣了。当然这不是说编程本身是一件无聊的事情，也不是说你以后也一定会这样觉得，这只是我个人在当前的感觉而已。

在这么久的旅程下来我的体会是：编程语言这东西并不重要，重要的是你用这些语言做的事情。事实上我一直知道这一点，不过以前我会周期性地被各种编程语言分神而忘记了这一点。现在我是永远不会忘记这一点了，你也不应该忘记这一点。

你学到和用到的编程语言并不重要。不要被围绕某一种语言的信仰把你扯进去，这只会让你忘掉了语言的真正目的，也就是作为你的工具来实现有趣的事情。

编程作为一项智力活动，是唯一一种能让你创建交互式艺术的艺术形式。你可以创建项目让别人使用，而且你可以间接地和使用者沟通。没有其他的艺术形式能做到如此程度的交互性。电影领着观众走向一个方向，绘画是不会动的。而代码却是双向互动的。

编程作为一项职业只是一般般有趣而已。编程可能是一份好工作，但如果你想赚更多的钱而且过得更快乐，你其实开一间快餐店就可以了。你最好的选择是将你的编程技术作为你其他职业的秘密武器。

技术公司里边会编程的人多到一毛钱一打，根本得不到什么尊敬。而在生物学、医药学、政府部门、社会学、物理学、数学等行业领域从事编程的人就能得到足够的尊敬，而且你可以使用这项技能在这些领域做出令人惊异的成就。

当然，所有的这些建议都是没啥意义的。如果你跟着这本书学习写软件而且觉得很喜欢这件事情的话，那你完全可以将其当作一门职业去追求。你应该继续深入拓展这个近五十年来极少有人探索过的奇异而美妙的智力工作领域。若能从中得到乐趣当然就更好了。

最后我要说的是学习创造软件的过程会改变你而让你与众不同。不是说更好或更坏，只是不同了。你也许会发现因为你会写软件而人们对你的态度有些怪异，也许会用“怪人”这样的词来形容你。也许你会发现因为你会戳穿他们的逻辑漏洞而他们开始讨厌和你争辩。甚至你可能会发现有人因为你懂得计算机怎么工作而觉得你是个讨厌的怪人。

对于这些我只有一个建议: 让他们去死吧。这个世界需要更多的怪人，他们知道东西是怎么工作的而且喜欢找到答案。当他们那样对你时，只要记住这是你的旅程，不是他们的。“与众不同”不是谁的错，告诉你“与众不同是一种错”的人只是嫉妒你掌握了他们做梦都不能想到的技

能而已。

你会编程。他们不会。这真他妈的酷。

下一步

现在还不能说你是一个程序员。这本书的目的相当于给你一个“编程棕带”。你已经了解了足够的编程基础，并且有能力阅读别的编程书籍了。读完这本书，你应该已经掌握了一些学习的方法，并且具备了该有的学习态度，这样你在阅读其他 Python 书籍时也许会更顺利，而且能学到更多东西。

我建议你看看这些项目，并尝试用他们创建一些什么：

> - `Learn Ruby The Hard Way` 你学习更多的编程语言,你将学习到更多关于编程的知识，所以试着学习Ruby。
> - `The Django Tutorial` 尝试使用 `Django Web Framework` 创建一个web应用。
> - `SciPy` `Dexy`：如果你在科学，数学，工程领域，如果你想写出很棒的论文，使用 SciPy 的代码
> - `PyGame` 看你能不能制作出一个带音效和图像的游戏
> - `Pandas` 用来做数据分析和处理
> - `Natural Language Tool Kit` 用来分析书面文本和写作比如垃圾邮件过滤器和聊天机器人。
> - `Requests` 了解HTTP客户端和WEB
> - `SimpleCV` 让你的计算机看到现实世界中的东西
> - `ScraPy` 网络爬虫
> - `Panda3D` 用来制作3D图画及游戏
> - `Kivy` 用来制作桌面和移动平台的用户界面。
> - `SciKit-Learn` 用来制作机器学习的应用
> - `Ren'Py` 用来做互动小说类的游戏，有点像在本书中你做过的游戏，但是这个是有图像的
> - `Learn C The Hard Way` 在你熟悉python语言之后，尝试用本书中的算法学习C语言，慢慢学 C 是不同的但很值得去学习的语言

选择一个上面的源代码,通读他们的所有说明手册和文档。当你阅读它的文档和代码的时候，输入所有的代码，并让代码运行起来。我就是这么做的。也是所有程序员的做法。阅读文档并不足够能使你学会它，你必须亲手实践。读完他们的说明手册和文档之后，尝试做一些小东西，任何东西都可以，即便是别人已经写过的。

Just understand anything you write will probably suck. That's alright though I suck at every programming language I first start using.？？ Nobody writes pure perfect gold when they're a beginner, and anyone who tells you they did is a huge liar.只要你明白你写的任何东西都将是吸引人的。每当我第一次开始使用一种语言编程的时候。。？没有人能在作为一个初学者的时候写出完美的代码，如果有人这么告诉你，那他一定是个大骗子。

如何学习其他编程语言

我将要教会你如何学习其他编程语言。本书的组织是基于我和很多其他程序员如何学习新的语言。我一般遵从一下流程：

> 1. 找一本关于这门语言的书或者其他说明资料
> 2. 通读这本书，练习输入这本书所有的代码，并保证他们能正常运行
> 3. 练习代码的同时仔细阅读这本书，并做笔记
> 4. 用这门语言实现一些小程序
> 5. 阅读别人用这门语言写代码，并尝试复制他们东西

本书中，我强迫你用很慢的速度一小部分一小部分的完成这个流程。其他的书中不一定是相同的方法，这意味着你要自己推断出我是如何让你进行这些步骤的去完成他们书中内容的。最好的办法是快速的阅读这本书，列出书中所有重要的代码段。把这个列表按章整理成一系列练习题，然后按顺序每次完成一个。

上面的流程同样适用于一些没有提供说明书给你的新技术。对于没有说明书的技术，你可以从网上搜索相关文档或源代码，并进行以上流程。

每学一门新语言，都会让你离更好的程序员更进一步，你学的越多，他们对你来说就越简单。通过你的第三或第四语言，你应该能够在一个星期内学会相似的语言，陌生的语言花费的时间要长一些。现在你已经学会了python，那么你就能通过比较快速的学会Ruby和js。这是因为许多语言有着相似的概念，一旦你学会一种，它们在其他语言里也是一样的。

你要记住的关于学习新语言的最后一件事情是：不要做一个愚蠢的观光者。愚蠢的人旅游到另一个国家，然后抱怨食物不像家里的食物。“在这个愚蠢的国家，为什么我不能获得一个更好的汉堡！”。当你学习一门新语言的时候，要坚信它不是无聊的，它只是跟之前的不同而已，拥抱它，你才能学得更好。

在你学习一种语言之后，不要成为一个以语言的方式做事情的奴隶。有时候，人们竟然使用语言做一些白痴的事情，仅仅是因为“我们一直是这么做的”。如果你喜欢你的风格并且你知道其他人都这样做，如果可以优化一些事情，那么打破这个规则。

我真的很享受学习新的编程语言。我认为自己是一个“程序员的人类学家”，并且认为使用这些语言的程序员只洞察到这门语言很小的一部分。我正在学习一门大家都用来在电脑上互相交流的语言，我发现它非常迷人。再说一次，我是一个奇怪的人，学习编程语言只是因为我想学。

享受它们！真的很有趣！

附录A：命令行教程

- 简介
- 安装和准备
- 路径, 文件夹, 名录 (pwd)
- 如果你迷路了
- 创建一个路径 (mkdir)
- 改变当前路径 (cd)
- 列出当前路径 (ls)
- 删除路径 (rmdir)
- 目录切换(pushd, popd)
- 生成一个空文件(Touch, New-Item)
- 复制文件 (cp)
- 移动文件 (mv)
- 查看文件 (less, MORE)
- 输出文件 (cat)
- 删除文件 (rm)
- 退出命令行 (exit)
- 下一步

附录A-简介

简介：使用shell命令行

这个附录是使用命令行的快速教程。作为快速教程，这部分内容不会像我其他的书一样详细。它仅仅是为了让你能够像一个真正的程序员一样使用的电脑。当你完成这个附录的学习，你将学会大部分shell用户每天使用的命令，你将明白基本的目录以及一些其他的概念。

对于附录内容，我给你的唯一意见是：

```
闭上嘴，练习输入每一个命令。
```

很抱歉这么说，但是这就是你必须要做的。如果你对命令行有非理性的恐惧心理， 征服它的唯一办法就是闭嘴，并与之斗争。

你并不是要毁掉你的电脑。 You are not going to be thrown into some jail at the bottom of Microsoft's Redmond campus. 你的朋友不会因为你变成一个书呆子而嘲笑你。所以，忽略你对命令行所有的愚蠢而奇怪的心理吧。

为什么这么说？因为如果你想学习编程的话，你必须先学习命令行的使用。编程是用编程语言来控制你计算机的高级方式。而命令行则是编程语言的婴儿小弟弟。学习命令行是在教你控制计算机语言。 Once you get past that, you can then move on to writing code and feeling like you actually own the hunk of metal you just bought. 当你通过了命令行的学习，你就可以继续编码，那种感觉就像你拥有了大块金属？？？

如何使用本附录

使用这个附录最好的办法是做到以下几点：

> - 给自己准备一个纸质笔记本和一支笔。
> - 从附录的开头开始，按照书中的要求完成每一项练习。
> - 当你读到一些你不明白的东西时，把他们记在笔记本上。留一点空间，这样你以后可以把答案写上。
> - 完成一个练习之后，退回去检查你在笔记本上记下的问题。尝试通过互联网或者你熟悉编程的朋友来获取答案。你也可以发邮件到 `help@learncodethehardway.org` 寻求帮助。

坚持做每一个练习，并写下你任何一个疑问，然后再想办法解决你的疑问。当你学完本附录之后，你会发现，你掌握的命令行知识比你想象的多得多。

你需要记下的东西

我提前警告你我会让你记住一些东西了。这是让你能掌握某些技能的最快的方式，但是对一些人来说，记忆可能是很痛苦的事情记忆对于学习任何东西都是很重要的技能，所以，你应该恐惧它。

这里是你如何记住东西的方法：

> - 告诉自己，你能记住它。不要试图寻找窍门或简单的方法，只要坐在那开始记忆就好。
> - 在索引卡片上写下你要记住的东西.把你要学的内容分成两部分，一半写在卡片的正面，一半写在背面。
> - 每天拿出15-30分钟时间，用做好的卡片训练自己，尝试回忆每一张卡片的内容。把任何你没有正确说出答案的卡片放到一边，针对这些卡片进行训练，直到你觉得厌烦，然后再尝试回忆所有的卡片，看你是否有所进步。
> - 睡觉之前，对你弄错了的卡在练习5分钟。

还有其他的方法，比如你可以把你要学习的内容写在一张纸上，然后将它贴在你浴室的墙上，当你洗漱的时候，你就可以不看着墙上的纸练习记忆这些内容，当你遇到问题的时候可以看一眼，刷新你的记忆。

如果你坚持每天都这样做，你应该能记住最多的事。 我想告诉你，练习记忆大约要一个星期到一个月。如果你这样做了，几乎所有的一切都变得更加容易和直观，这就是记忆的目的。这并不是教你什么抽象的概念，而是一些根深蒂固的基础知识，你不需要思考它们就能脱口而出的知识。如果你记住了这些基础知识，它们就不会再是影响你学习更高级内容的拦路虎了。

附录A-练习1：安装

本附录中，你需要完成3件事：

> - 用你的终端做一些事情 (command line, Terminal, PowerShell).
> - 了解你做过的事情.
> - 自己多练习.

在第一个练习中，你将学会如何打开你的终端并使用其工作，这样你才能完成本附录后面部分的学习。

做到这些

让你的终端保持工作状态，这样你就可以快速访问它，并了解它的工作原理。

Mac OSX

在Mac OSX系统上，你应该

> - 按住 `command` 键，并敲空格键。
> - 屏幕顶部会弹出一个蓝色的“搜索框”。
> - 输入“terminal”。
> - 点击终端应用程序，这个程序的图标看起来有点像一个黑盒子。
> - 终端就打开了。
> - 现在你可以在你的dock中看到你终端的那个图表，选中它右键选择选项-->保留，这样你的终端就会一直保留在dock中了。

你现在已经打开了你的终端，并将它放在你dock中，这样你下次可以快速的打开它。

Linux

如果你用的是Linux系统的话，我假设你知道如何打开你的终端。通过菜单窗口管理器查找叫做shell或者terminal的应用。

Windows

在windows系统中，我们要使用PowerShell。人们常用一个名为 `cmd.exe` 的程序协同工作，但是它并不像PowerShell好用。如果你有Windows7或以上版本,这样做：

> - 单击开始菜单
> - 在“搜索程序和文件”中输入“ powershell”。
> - 敲回车

如果你没有Windows 7，你应该考虑升级你的系统。如果你坚持不想升级，你可以尝试从微软的下载中心安装它。网上搜索一下，找到"powershell下载"。 安装适合你电脑的版本，虽然我没有Windows XP，但我仍希望PowerShell的体验是一样的。

你应该学到的

你已经学会如何打开你的终端了，现在你可以继续学习本附录的其余部分了。

> **NOTE:**如果你有一些熟悉Linux系统的朋友，当他告诉用一些其他的东西替代Bash的时候，忽略他的话。我正在教你使用bash。就是这样！即使他声称，ZSH能让你提升30个IQ值甚至更多，忽视他！你的目标是在当前级别获得足够的能力，所以你用什么shell没有什么关系。接下来的警告是远离IRC或其他有黑色出没定的地方。他们认为破坏你的电脑很有趣。 命令`rm -rf /`是一个最经典的你永远也不能使用的命令。躲开他们。如果你需要帮助，确保你是从你信任的地方获得答案，而不是从互联网上随便哪个白痴哪里得到帮助。

更多练习

这节练习有一个很大的“更多练习”部分。其他的练习是没有这么复杂的更多练习的， 但是，对于本附录的其余部分，我需要你用的大脑做一些记忆的事情。相信我：这会让以后的事情如丝般柔滑！

Linux/Mac OSX

给下表中的命令创建索引卡片，把命令名称写在卡片的左侧，把命令的定义或功能写在右侧。当你继续本附录中的其他课程时，也要每天抽出时间练习它们。

pwd: 打印当前工作目录

hostname: 获取我的计算机的网络名称

mkdir: 创建目录

cd: 更改目录

ls: 列出目录下的文件

rmdir: 删除目录

pushd: push directory

popd: pop directory

cp: 复制文件或目录

mv: 移动/重命名文件或目录

less: 按页查看文件

cat: 输出整个文件

xargs: 执行参数

find: 查找文件

grep: 查找文件里面的东西

man: 阅读帮助手册

apropos: find what man page is appropriate

env: 查看计算机环境

echo: 输出一些参数

export: 设置一个新的环境变量

exit: 退出终端

sudo: 危险! 拥有超级用户权限!

Windows

如果你用的是windows系统，你要熟记以下命令：

pwd: 打印当前工作目录

hostname: 获取我的计算机的网络名称

mkdir: 创建目录

cd: 更改目录

ls: 列出目录下的文件

rmdir: 删除目录

pushd: push directory

popd: pop directory

cp: 复制文件或目录

robocopy: 更强大的复制

mv: 移动/重命名文件或目录

more: 按页查看文件

type: 输出整个文件

forfiles: 对大量文件执行一个操作

dir -r: 查找文件

select-string: 查找文件里面的东西

help: 阅读帮助手册

helpctr: find what man page is appropriate

echo: 输出一些参数

set: 设置一个新的环境变量

exit: 退出终端

runas: 危险! 拥有超级用户权限!

练习、练习、练习! 练习到你看到一个词能马上说出它的功能。然后倒着练习，你看到一个功能，知道用什么命令实现它。

附录A-练习2：路径, 文件夹, 名录 (pwd)

在这个练习中，你将学会使用 `pwd` 命令打印工作目录。

做到这些

我要教会你如何阅读这些我告诉你的“会话”。你不必输入我在这里列出的一切，练习输入其中的一部分就好：

> - 你不需要输入 `$` (Unix) 或 `>` (Windows). 这只是我向你展示你在我的会话中能看到什么。
> - 你在 `$` 或 `>` 之后输入命令，然后敲回车。所以如果我写的是 `$ pwd` 你只需要输入 `pwd` 并敲回车。
> - 紧跟着你可以在另一个 `$` 或 `>` 符号后面看到结果输出。这些内容是输出，你应该看到了相同的输出

让我们使用一个简单的命令，你可以得到这样的窍门：

Linux/OSX

```
$ pwd
/Users/zedshaw
$
```

Windows

```
PS C:\Users\zed> pwd

Path
----
C:\Users\zed

PS C:\Users\zed>
```

> **NOTE:**在本附录中，我需要节省空间，这样你可以更专注于命令的细节。为了达到这个目的，我会删除提示信息的第一部分 (上面例子中的 `PS C:\Users\zed` 部分) ，并只留下符号 `>` 部分。这意味着你的提示信息不会长的一模一样，不过不用担心这些。

记住从现在开始，在提示部分我只会留下 `>` 。

对于Unix的提示，我也做了相同处理，但Unix的提示于windows不同，大多数习惯用 `$` 。

你应该学到的

你的提示信息与我的不同。你的提示信息可能在`$`之前有你的名字或者你电脑的名字。在Windows系统上，应该也是不同的。关键是你看到的模式：

> - 有一个提示。
> - 你可以在这里输入命令，这节中的例子是输入`pwd`。
> - 可以打印输出一些内容。
> - 重复以上内容。

你已经学到了`pwd`是做什么的，它的意识是“打印当前工作目录”。什么是一个目录？就是一个文件夹。文件夹和目录是一回事，它们可以互换使用。 当你在你的计算机上以图形方式打开文件浏览器查找文件的时候，你可以浏览你的整个文件夹。这些文件夹与我们所说的目录就是完全相同的东西。

更多练习

> - 输入210遍`pwd`，每次都要说 "打印当前工作目录"。
> - 写下这个命令输出给你的路径。使用你的图形文件浏览器找到它。
> - 还不够，你要再输入20遍，每一遍都要大声的朗读出来。

附录A-练习3：如果你迷路了

当你经历了这些说明，你可能会迷失自己。你可能不知道你在哪里或者再哪一个文件，而且也不知道该如何继续。为了解决这个问题，我要叫你一个命令，这个命令可以防止你在文件目录中迷失方向。

当你迷路的时候，最可能的原因是你执行了某些命令，但你不知道当前你在哪个目录下。此时，你应该输入命令 `pwd` 来打印当前目录。这个命令告诉你你在哪里。

接下来要做的事情就是你需要一种方式回到你的home目录。输入 `cd ~` 你将会回到你的home目录

也就是说，当你迷路的时候，你可以输入:

```
pwd
cd ~
```

第一个命令`pwd` 告诉你你现在在哪来。第二个命令 `cd ~` 带你回到你的home目录。

做到这些

现在，找到你在哪来，然后通过使用 `pwd` 和 `cd ~`命令回到你的home目录。 这能保证你总是在正确的地方。

你应该学到的

如果你不知道你现在位于什么位置，你应该知道怎样回到home目录。

附录A-练习4：创建一个路径 (mkdir)

这节练习，你将学习如果使用`mkdir`命令创建一个新的目录（文件夹）。

做到这些

记住！ 你要先回到你的home目录！在你做这节练习之前，先执行`pwd`和`cd ~`操作。 在你做本附录的所有练习之前，都先回到home目录！

Linux/OSX

```
$ pwd
$ cd ~
$ mkdir temp
$ mkdir temp/stuff
$ mkdir temp/stuff/things
$ mkdir -p temp/stuff/things/frank/joe/alex/john
$
```

Windows

```
> pwd
> cd ~
> mkdir temp

    Directory: C:\Users\zed

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:02 AM            temp

> mkdir temp/stuff

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:02 AM            stuff

> mkdir temp/stuff/things

    Directory: C:\Users\zed\temp\stuff

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:03 AM            things

> mkdir temp/stuff/things/frank/joe/alex/john

    Directory: C:\Users\zed\temp\stuff\things\frank\joe\alex

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:03 AM            john

>
```

这是我唯一一次给你列出 `pwd` 和 `cd ~` 命令。 它们预计会在每次练习中使用。随时使用它们。

你应该学到的

现在我们已经开始学习输入多个命令。这些都是可以运行`mkdir`的方式。 `mkdir` 是做什么的？它能创建目录。你为什么问这个？你应该用你的索引卡片练习记忆这些命令。如果你不知道" `mkdir` 创建目录"，那你应该继续使用索引卡练习。

创建一个目录是什么意思？你也可以把目录叫做文件夹。他们是同一种东西。 所有你上面做的事情是创建目录内的目录。这也可以叫做“路径”这是一个跟你说 "第一temp, 然后stuff, 接下来things，这就是我想要的"的方式。这在你的计算机硬盘上就是一系列树形结构的文件夹。

> **NOTE:**在本附录中，我在所有路径中使用 `/` (slash) 字符，因为现在他们在所有计算机上都能正常工作。然而， Windows系统使用者需要知道，你们也可以使用`\` (backslash)。

更多练习

> - "path" 的概念可能会使你迷惑。别担心。我们将用他们做更多的练习，然后你就会理解了。
> - 在temp目录下创建20个其他的目录，这20个目录要在不同的层级上。通过你的图形文件浏览器看看你创建的目录。
> - 试试看创建一个名字中带有空格且被双引号包起来的目录：`mkdir "I Have Fun"`
> - 如果temp目录已经存在，那么你将得到一个错误。使用 `cd` 改变当前工作目录，然后再尝试创建不同目录。在Windows中，桌面是一个好地方。

附录A-练习5：改变当前路径 (cd)

这节练习中，你将学习如何使用 `cd` 命令从一个目录切换到另一个。

做到这些

我打算再一次给你解释这些会话的内容：

> - 你不需要输入 `$` (Unix) 或 `>` (Windows).
> - 你输入stuff然后敲回车。如果我是 `$ cd temp` 你只需要输入 `cd temp` 然后回车。
> - 输出会在你按下回车键之后展现，跟在另一个 `$` 或 `>` 提示符之后.
> - 永远先回到home目录! 执行 `pwd` 和 `cd ~` 。

Linux/OSX

```
$ cd temp
$ pwd
~/temp
$ cd stuff
$ pwd
~/temp/stuff
$ cd things
$ pwd
~/temp/stuff/things
$ cd frank/
$ pwd
~/temp/stuff/things/frank
$ cd joe/
$ pwd
~/temp/stuff/things/frank/joe
$ cd alex/
$ pwd
~/temp/stuff/things/frank/joe/alex
$ cd john/
$ pwd
~/temp/stuff/things/frank/joe/alex/john
$ cd ..
$ cd ..
$ pwd
~/temp/stuff/things/frank/joe
$ cd ..
$ cd ..
$ pwd
~/temp/stuff/things
$ cd ../../..
$ pwd
~/
$ cd temp/stuff/things/frank/joe/alex/john
$ pwd
~/temp/stuff/things/frank/joe/alex/john
$ cd ../../../../../../../
$ pwd
~/
$
```

Windows

```
> cd temp
> pwd

Path
----
C:\Users\zed\temp

> cd stuff
> pwd

Path
----
C:\Users\zed\temp\stuff

> cd things
> pwd

Path
----
C:\Users\zed\temp\stuff\things

> cd frank
> pwd

Path
----
C:\Users\zed\temp\stuff\things\frank

> cd joe
> pwd

Path
----
C:\Users\zed\temp\stuff\things\frank\joe

> cd alex
> pwd

Path
----
C:\Users\zed\temp\stuff\things\frank\joe\alex

> cd john
> pwd

Path
----
C:\Users\zed\temp\stuff\things\frank\joe\alex\john

> cd ..
> cd ..
> cd ..
> pwd

Path
----
C:\Users\zed\temp\stuff\things\frank

> cd ../..
> pwd

Path
----
C:\Users\zed\temp\stuff

> cd ..
> cd ..
> cd temp/stuff/things/frank/joe/alex/john
```

```
> cd ../../../../../../../
> pwd

Path
----
C:\Users\zed

>
```

你应该学到的

上节练习中你已经创建了所有的目录，你现在只需要使用`cd`命令，就能实现在它们之间进行切换。在上面我的会话中，我也使用`pwd`来检查我在哪里，所以一定记得不要输入命令`pwd`所输出的内容。比如，在第3行中，你看到`~/temp`，但是它是上面一个`pwd`命令的输出。所以不要输入这行。

你应该看到我如何使用`..`来移动到上一层目录的。

更多练习

学习在计算机上使用命令行模式(CLI)与图形用户界面(GUI)的一个非常重要的部分弄清楚他们是如何协同工作的。当我刚开始使用电脑时，是没有GUI的，我要做的一切都是用DOS提示符（命令行）来实现的.后来,当电脑变得足够强大,每个人都可以通过GUI操作电脑的时候,GUI窗口和CLI目录文件夹协同使用对我来说是很简单的。

今天的大多数人，并不理解CLI、路径和目录的概念。实际上，很难教会他们理解这些，唯一的学习方式是给你不断的使用CLI，直到有一天你点击你在GUI中做的东西，而它能出现在CLI中。

做到这些的方法是你花一些时间找到你的GUI文件浏览器，然后通过你的CLI进入文件浏览器。这些是你下一步要做的事情：

> - 使用一个命令进入`joe`目录。
> - 使用一个命令回到`temp`目录，但不能使用上面例子中的命令。
> - 找到使用一个命令回到 "home 目录" 的方法。
> - 进入你的文件目录，然后使用你的GUI文件浏览器找到这个目录。
> - 进入你的下载目录，然后使用你的GUI文件浏览器找到这个目录。
> - 使用你的GUI文件浏览器找到另一个目录，然后使用`cd`进入这个目录。
> - 还记不记得你用引号包围一个名字中有空格的目录？你可以使用任何命令这么做。比如，你有一个目录叫做`I Have Fun`，那你可以执行: `cd "I Have Fun"`

附录A-练习6：列出当前路径 (ls)

这节练习中你将学习如何使用 `ls` 命令列出一个目录下的所有内容。

做到这些

开始之前，确认你已经 `cd` 回到temp的上一层目录。.如果你不知道你在哪里，使用 `pwd` 找到你的位置，然后移动到正确的目录下。

Linux/OSX

```
$ cd temp
$ ls
stuff
$ cd stuff
$ ls
things
$ cd things
$ ls
frank
$ cd frank
$ ls
joe
$ cd joe
$ ls
alex
$ cd alex
$ ls
$ cd john
$ ls
$ cd ..
$ ls
john
$ cd ../../../
$ ls
frank
$ cd ../../
$ ls
stuff
$
```

Windows

```
> cd temp
> ls

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:03 AM            stuff

> cd stuff
> ls

    Directory: C:\Users\zed\temp\stuff
```

```
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:03 AM            things

> cd things
> ls

    Directory: C:\Users\zed\temp\stuff\things

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:03 AM            frank

> cd frank
> ls

    Directory: C:\Users\zed\temp\stuff\things\frank

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:03 AM            joe

> cd joe
> ls

    Directory: C:\Users\zed\temp\stuff\things\frank\joe

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:03 AM            alex

> cd alex
> ls

    Directory: C:\Users\zed\temp\stuff\things\frank\joe\alex

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:03 AM            john

> cd john
> ls
> cd ..
> ls

    Directory: C:\Users\zed\temp\stuff\things\frank\joe\alex

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:03 AM            john

> cd ..
> ls

    Directory: C:\Users\zed\temp\stuff\things\frank\joe

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:03 AM            alex

> cd ../../..
> ls

    Directory: C:\Users\zed\temp\stuff

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:03 AM            things

> cd ..
> ls
```

```
    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:03 AM            stuff

>
```

你应该学到的

`ls`能列出你当前目录中的所有内容。你可以看到我用`cd`切换到不同的目录下，然后列出该目录下的所有内容，这样，我就知道我接下来要到哪个目录中去了。

`ls`命令有很多的命令选项，当我们学习`help`命令之后，你将会学习如何得到这些命令选项的帮助信息。

更多练习

> - 输入这里的每一个命令！你必须亲手输入这些命令来学习他们。只是读他们并不够。
> - 在Unix中, 在`temp`目录下试试命令`ls -lR`。
> - 在Windows中也可以试试`dir -R`.
> - 使用`cd`进入到你计算机上的其他目录，然后使用`ls`看看当前目录里有什么。
> - 将你遇到的新问题更新到你的笔记本中。我知道你可能已经有一些问题了，因为我并没有覆盖这些命令的所有点。
> - 记住，如果你迷路了，使用`ls`和`pwd`找到你在哪里，然后使用`cd`去到你要去的目录。

附录A-练习7：删除路径 (rmdir)

这节练习中，你将学习如何删除一个空目录。

做到这些

Linux/OSX

```
$ cd temp
$ ls
stuff
$ cd stuff/things/frank/joe/alex/john/
$ cd ..
$ rmdir john
$ cd ..
$ rmdir alex
$ cd ..
$ ls
joe
$ rmdir joe
$ cd ..
$ ls
frank
$ rmdir frank
$ cd ..
$ ls
things
$ rmdir things
$ cd ..
$ ls
stuff
$ rmdir stuff
$ pwd
~/temp
$
```

> **Warning:**如果你在Mac OSX上尝试执行rmdir命令，即使你确认这个目录是空的，但是计算机仍拒绝删除该目录，那么实际上应该是有一个名为 `.DS_Store` 的文件。这种情况下，你可以输入 `rm -rf <dir>` 来执行删除操作(将 `<dir>` 替换为你要删除的目录名)。

Windows

```
> cd temp
> ls

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:03 AM            stuff

> cd stuff/things/frank/joe/alex/john/
> cd ..
> rmdir john
> cd ..
> rmdir alex
> cd ..
> rmdir joe
> cd ..
> rmdir frank
> cd ..
> ls

    Directory: C:\Users\zed\temp\stuff

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:14 AM            things

> rmdir things
> cd ..
> ls

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/17/2011   9:14 AM            stuff

> rmdir stuff
> pwd

Path
----
C:\Users\zed\temp

> cd ..
>
```

你应该学到的

现在我把这些命令混合起来了，所以你需要加倍注意，并确保你输入了正确的命令。每次你犯一个错误，都是因为你没有仔细看。如果你发现自己犯了不少错误，那么休息以下，或者退出练习一天。你总是需要明天再试一次的。

这个例子中，你学习了如何删除一个目录。很简单，你只需要在该目录的上一级目录，输入 `rmdir <dir>` , 把 `<dir>` 替换成你要删除的目录名。

更多练习

> - 创建20个以上的目标，再把他们都删除。
> - 创建一个深度为10的单一路径的目录，然后每次删除其中的一个。
> - 如果你尝试删除一个不为空的目录，你会得到一个错误。在后面的练习中我会告诉你如何删除这样的目录。

附录A-练习8：目录切换(pushd, popd)

这节练习中，你将学习如何使用 `pushd` 实现保存你的当前位置，并去一个新的位置，然后您将学习如何使用 `popd` 恢复保存位置。

做到这些

Linux/OSX

```
$ cd temp
$ mkdir -p i/like/icecream
$ pushd i/like/icecream
~/temp/i/like/icecream ~/temp
$ popd
~/temp
$ pwd
~/temp
$ pushd i/like
~/temp/i/like ~/temp
$ pwd
~/temp/i/like
$ pushd icecream
~/temp/i/like/icecream ~/temp/i/like ~/temp
$ pwd
~/temp/i/like/icecream
$ popd
~/temp/i/like ~/temp
$ pwd
~/temp/i/like
$ popd
~/temp
$ pushd i/like/icecream
~/temp/i/like/icecream ~/temp
$ pushd
~/temp ~/temp/i/like/icecream
$ pwd
~/temp
$ pushd
~/temp/i/like/icecream ~/temp
$ pwd
~/temp/i/like/icecream
$
```

Windows

```
> cd temp
> mkdir -p i/like/icecream

    Directory: C:\Users\zed\temp\i\like

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/20/2011  11:05 AM            icecream

> pushd i/like/icecream
> popd
> pwd

Path
----
C:\Users\zed\temp

> pushd i/like
> pwd

Path
----
C:\Users\zed\temp\i\like

> pushd icecream
> pwd

Path
----
C:\Users\zed\temp\i\like\icecream

> popd
> pwd

Path
----
C:\Users\zed\temp\i\like

> popd
>
```

你应该学到的

你使用这些命令进入了程序员的领土，这些命令是如何方便使用，我一定要教会你使用。这些命令让你暂时去到不同的目录，然后再回来，实现两个目录之间的轻松切换。

`pushd` 命令把你当前的目录放到一个list中，然后切换到另一个目录。就好像说"保存我在哪里，然后去下一个地方"

`popd` 命令把你之前存储的目录弹出来，然后带你回到那个目录。

最后， 在Unix上，如果你不带任何参数运行 `pushd` ，将会在当前目录以及你之前最后一次保存的目录之间进行切换。这是来回切换两个目录的方法，但它在Powershell中不起作用。

更多练习

> - 使用这两个命令在你计算机上的所有目录中来回切换。
> - 删除目录 `i/like/icecream` 再自己创建几个目录，在你新创建的目录间进行切换。
> - 给自己解释下 `pushd` 和 `popd` 命令的输出。注意下，它是如何像一个堆栈一样工作的。
> - 你已经知道这些, 但请记住 `mkdir -p` 将会创建整个路径，即使其中的目录都不存在。这是我在这节练习中首先要做的事情。

附录A-练习9：生成一个空文件(Touch, New-Item)

这节练习中，你将学习使用 `touch` (Windows中是 `new-item`)命令创建一个空文件。

做到这些

Linux/OSX

```
$ cd temp
$ touch iamcool.txt
$ ls
iamcool.txt
$
```

Windows

```
> cd temp
> New-Item iamcool.txt -type file
> ls

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        12/17/2011   9:03 AM            iamcool.txt

>
```

你应该学到的

你已经学会了如何创建一个空文件。在Unix中使用 `touch` 命令,它也改变了文件的修改时间。我只用这个命令还创建空文件。在Windows上，没有这命令，所以你要学习如何使用 `New-Item` 命令，这个命令可以创建空文件也可以创建创建一个新目录。

更多练习

> - Unix: 创建一个目录，进入它并在里面创建一个文件。然后回到他的上一级目录，执行 `rmdir` 命令，你将会得到一个错误。尝试想一想为什么会有这个错。
> - Windows: 做同样的事情，但你不会看到这个错误。你将会得到一个提示，询问你是否真的要删除该目录。

附录A-练习10：复制文件 (cp)

这节练习中你将使用 `cp` 命令从一个位置复制一个文件到另一个位置。

做到这些

Linux/OSX

```
$ cd temp
$ cp iamcool.txt neat.txt
$ ls
iamcool.txt neat.txt
$ cp neat.txt awesome.txt
$ ls
awesome.txt iamcool.txt neat.txt
$ cp awesome.txt thefourthfile.txt
$ ls
awesome.txt  iamcool.txt  neat.txt  thefourthfile.txt
$ mkdir something
$ cp awesome.txt something/
$ ls
awesome.txt iamcool.txt  neat.txt  something  thefourthfile.txt
$ ls something/
awesome.txt
$ cp -r something newplace
$ ls newplace/
awesome.txt
$
```

Windows

```
> cd temp
> cp iamcool.txt neat.txt
> ls

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        12/22/2011   4:49 PM          0 iamcool.txt
-a---        12/22/2011   4:49 PM          0 neat.txt

> cp neat.txt awesome.txt
> ls

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        12/22/2011   4:49 PM          0 awesome.txt
-a---        12/22/2011   4:49 PM          0 iamcool.txt
-a---        12/22/2011   4:49 PM          0 neat.txt

> cp awesome.txt thefourthfile.txt
> ls

    Directory: C:\Users\zed\temp
```

```
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        12/22/2011   4:49 PM          0 awesome.txt
-a---        12/22/2011   4:49 PM          0 iamcool.txt
-a---        12/22/2011   4:49 PM          0 neat.txt
-a---        12/22/2011   4:49 PM          0 thefourthfile.txt

> mkdir something

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/22/2011   4:52 PM            something

> cp awesome.txt something/
> ls

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/22/2011   4:52 PM            something
-a---        12/22/2011   4:49 PM          0 awesome.txt
-a---        12/22/2011   4:49 PM          0 iamcool.txt
-a---        12/22/2011   4:49 PM          0 neat.txt
-a---        12/22/2011   4:49 PM          0 thefourthfile.txt

> ls something

    Directory: C:\Users\zed\temp\something

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        12/22/2011   4:49 PM          0 awesome.txt

> cp -recurse something newplace
> ls newplace

    Directory: C:\Users\zed\temp\newplace

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        12/22/2011   4:49 PM          0 awesome.txt

>
```

你应该学到的

现在你会复制文件了。这是简单的只获取一个文件，并复制到一个新文件。在这个练习中，我也创建了一个新目录，并将文件复制到该目录中。

我要告诉你一个关于程序员和系统管理员的秘密了。他们很懒，我也很懒，我的朋友们也很懒。这就是为什么我们要使用电脑。我们喜欢让电脑为我们做无聊的事情。在目前的练习中，为了使你了解这些命令，你需要重复键入这些枯燥的命令，但通常都不是这样的。通常，如果你发现自己正在做一些无聊或重复的事情，有可能已经有程序员找到更容易做到的方法了。只是你不知道这件事。

关于程序员的另一个秘密是，他们并不像你想象的那样聪明。如果你过多的思考要输入的内容，那你肯呢过就搞错了。相反，想象一下对你来说一个命令的名字是什么。可能是一个名字或者一些类似你认为的缩写。如果你仍然无法搞清楚，那么问问周围的人或者上网找找答案。但愿这不是跟ROBOCOPY一样愚蠢的东西。

更多练习

> - 使用 `cp -r`命令，复制一个包含文件的目录。
> - 复制一个文件到你的home目录或桌面。
> - 在你的GUI中找到这些文件，并用文本编辑器打开它们。
> - 请注意，为什么有时候我会在一个目录的结尾用一个`/` (slash)？这可以确保该文件确实是一个目录，如果没有这个目录，我就会得到一个错误。

附录A-练习11：移动文件 (mv)

这节练习中，你将学习使用 `mv` 命令把文件从一个位置移动另一个位置。

做到这些

Linux/OSX

```
$ cd temp
$ mv awesome.txt uncool.txt
$ ls
newplace    uncool.txt
$ mv newplace oldplace
$ ls
oldplace    uncool.txt
$ mv oldplace newplace
$ ls
newplace   uncool.txt
$
```

Windows

```
> cd temp
> mv awesome.txt uncool.txt
> ls

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/22/2011   4:52 PM            newplace
d----        12/22/2011   4:52 PM            something
-a---        12/22/2011   4:49 PM          0 iamcool.txt
-a---        12/22/2011   4:49 PM          0 neat.txt
-a---        12/22/2011   4:49 PM          0 thefourthfile.txt
-a---        12/22/2011   4:49 PM          0 uncool.txt

> mv newplace oldplace
> ls

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/22/2011   4:52 PM            oldplace
d----        12/22/2011   4:52 PM            something
-a---        12/22/2011   4:49 PM          0 iamcool.txt
-a---        12/22/2011   4:49 PM          0 neat.txt
-a---        12/22/2011   4:49 PM          0 thefourthfile.txt
-a---        12/22/2011   4:49 PM          0 uncool.txt

> mv oldplace newplace
> ls newplace

    Directory: C:\Users\zed\temp\newplace

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        12/22/2011   4:49 PM          0 awesome.txt

> ls

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/22/2011   4:52 PM            newplace
d----        12/22/2011   4:52 PM            something
-a---        12/22/2011   4:49 PM          0 iamcool.txt
-a---        12/22/2011   4:49 PM          0 neat.txt
-a---        12/22/2011   4:49 PM          0 thefourthfile.txt
-a---        12/22/2011   4:49 PM          0 uncool.txt

>
```

你应该学到的

移动文件，或者重命名文件。这很简单： 给出就名字，然后新名字。

更多练习

> - 将 `newplace` 目录中的一个文件移动到另一个目录，在移动回来。

附录A-练习12：查看文件 (less, MORE)

为了做这个练习，你要用你已经知道的命令做一些工作。你还需要一个文本编辑器，可以编写纯文本（.txt）文件。这里是你要做的：

> - 打开文本编辑器，并输入一些内容到一个新文件。
> - 保存文件到你的桌面，并将其命名为 `test.txt` 。
> - 在你的命令行中，使用你已知的命令复制这个文件到 `temp` 目录下。

当你完成这些准备工作，开始完成这个练习吧。

做到这些

Linux/OSX

```
$ less test.txt
[displays file here]
$
```

就是这样，输入 `q` 退出 `less` 命令。

Windows

```
> more test.txt
[displays file here]
>
```

> **NOTE:**上面例子中的输出，我使用 `[displays file here]` 代替程序显示的内容。 我这样做的意思是说"显示你这个程序的输出太复杂了，所以只需要在这里插入你在你电脑上看到的内容，假装我展示给你看了。"你的屏幕实际上不会这样显示。

你应该学到的

这是查看文件内容的一种方法。它很有用，因为如果这个文件有很多行，它将按页展现文件的内容，这样每次都只有一屏内容是可见的。在更多练习部分，你会做更多关于这个的练习。

更多练习

> - 再次打开你的文件，然后复制粘贴文件的文本内容，使文件有50-100行。
> - 将文件复制到你的`temp`目录中。
> - 现在再做一次这个练习。在Unix中，你可以使用空格键以及字母键 `w` 来向下或向上查看。方向键也适用。在Windows上，只能通过空格键翻页。
> - 看看你创建的空文件。
> - `cp` 命令将会覆盖已经存在的文件，所以复制文件的时候一定要当心。

附录A-练习13：输出文件 (cat)

你要为这节练习做更多的准备工作，你要习惯于在一个程序中创建文件，然后通过命令行访问这个文件。使用上节练习中用过的文本编辑器，新建一个叫做 `test2.txt` 的文件，这一次直接将他保存在你的 `temp` 目录中.

做到这些

Linux/OSX

```
$ less test2.txt
[displays file here]
$ cat test2.txt
I am a fun guy.
Don't you know why?
Because I make poems,
that make babies cry.
$ cat test.txt
Hi there this is cool.
$
```

Windows

```
> more test2.txt
[displays file here]
> cat test2.txt
I am a fun guy.
Don't you know why?
Because I make poems,
that make babies cry.
> cat test.txt
Hi there this is cool.
>
```

请记住，当我说 `[displays file here]` 的时候，我只是缩写了命令的真实输出，所以我没有给你展示确切的一切。

你应该学到的

你喜欢我的诗吗？你已经知道第一个命令，我只是让你检查你的文件是否存在。然后你使用 `cat` 输出该文件到屏幕上。这个命令会将整个文件内容不分页不间断的输出到屏幕上。为了证明这一点，我需要你同样使用该命令输出l `test.txt` ，这个文件会输出一大堆文字。

更多练习

> - 创建更多的文本文件，并使用`cat`命令输出文件内容。
> - Unix: 尝试命令`cat test.txt test2.txt` 看看它做了什么。
> - Windows: 尝试命令 `cat test.txt,test2.txt` 看看它做了什么。

附录A-练习14：删除文件 (rm)

这节练习中，你将学会如何使用 `rm` 命令删除一个文件。

做到这些

Linux

```
$ cd temp
$ ls
uncool.txt  iamcool.txt  neat.txt  something  thefourthfile.txt
$ rm uncool.txt
$ ls
iamcool.txt  neat.txt  something  thefourthfile.txt
$ rm iamcool.txt neat.txt thefourthfile.txt
$ ls
something
$ cp -r something newplace
$
$ rm something/awesome.txt
$ rmdir something
$ rm -rf newplace
$ ls
$
```

Windows

```
> cd temp
> ls

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/22/2011   4:52 PM            newplace
d----        12/22/2011   4:52 PM            something
-a---        12/22/2011   4:49 PM          0 iamcool.txt
-a---        12/22/2011   4:49 PM          0 neat.txt
-a---        12/22/2011   4:49 PM          0 thefourthfile.txt
-a---        12/22/2011   4:49 PM          0 uncool.txt

> rm uncool.txt
> ls

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/22/2011   4:52 PM            newplace
d----        12/22/2011   4:52 PM            something
-a---        12/22/2011   4:49 PM          0 iamcool.txt
-a---        12/22/2011   4:49 PM          0 neat.txt
-a---        12/22/2011   4:49 PM          0 thefourthfile.txt

> rm iamcool.txt
> rm neat.txt
> rm thefourthfile.txt
> ls

    Directory: C:\Users\zed\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        12/22/2011   4:52 PM            newplace
d----        12/22/2011   4:52 PM            something

> cp -r something newplace
> rm something/awesome.txt
> rmdir something
> rm -r newplace
> ls
>
```

你应该学到的

这里我们清理了之前练习中的所有文件。还记得我让你尝试使用 `rmdir` 删除一个不为空的目录吗？那个操作失败了因为你无法删除包含文件在内的目录。要做到这一点，你需要删除文件，或者递归删除所有的内容。这是你要在本节练习结尾要做的事情。

更多练习

> - 清理从开始练习到现在所有 `temp` 目录下的文件。
> - 在你的笔记本上写下递归删除文件时一定要小心。

附录A-练习15:退出命令行 (exit)

做到这些

Linux/OSX

```
$ exit
```

Windows

```
> exit
```

你应该学到的

最后一个练习是学会如何退出一个终端命令行。这个非常简单，但是我还是希望你能多做些练习。

更多练习

在你最后的练习题中，我会告诉你如何使用帮助系统，使用帮助系统可以查看和研究并学会使用更多的命令行命令。

Unix中你需要自己研究的命令:

> - xargs
> - sudo
> - chmod
> - chown

Windows 中你需要自己研究的命令:

> - forfiles
> - runas
> - attrib
> - icacls

找出这些命令是干什么的，并练习使用它们，然后把他们加入到你的索引卡中。

附录A-下一步

命令行学习下一步

你已经完成了这本快速教程。此时，你应该能勉强算是一个shell用户了。 但是仍然还有一大部分的操作是你所不知道的，我想在最后给你一些需要研究的内容。

Unix Bash 参考（Unix/Linux环境）

在Unix环境下你所用的shell叫做 `Bash` 。它不是最大的shell程序，但是它无所不在，而且又很多的功能，所以，学习Bash是一个好的开始。下面是几个学习Bash可以参考的链接：

Bash 使用小技巧: `http://cli.learncodethehardway.org/bash_cheat_sheet.pdf`

参考手册: `http://www.gnu.org/software/bash/manual/bashref.html`

PowerShell 参考（windows环境）

在Windows环境下，通常只有 `PowerShell` ，下面是一系列对你有用的文档：

用户手册: `http://technet.microsoft.com/en-us/library/ee221100.aspx`

使用技巧: `http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=7097`

PowerShell大师: `http://powershell.com/cs/blogs/ebook/default.aspx`